VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Nástroj pro modelování výpočetních pipeline

Tool for Modeling Computing Pipelines

Bc. Martin Huber

Diplomová práce

Vedoucí práce: Ing. Jan Kožusznik, Ph.D.

Ostrava, 2021

Abstrakt

Tato diplomová práce si klade za cíl vytvořit nástroj pro vizuální modelování výpočetních Snakemake pipelines a implementovat jej do systému pro jejich vzdálenou správu. Nejprve je čtenář seznámen s tímto systémem a nástrojem Snakemake. Poté je provedena rešerše v oblasti existujících nástrojů řešících danou problematiku. Za účelem realizace řešení jsou analyzovány různé přístupy a následně je navrženo konkrétní řešení skrze rozšíření webové aplikace Elsa Designer. Řešení je pak podle tohoto návrhu implementováno a integrováno do systému pro vzdálenou správu pipelines. K tomuto řešení jsou následně vytvořeny automatické testy, jejichž spuštění je zdokumentováno a vyhodnoceno.

Klíčová slova

Snakemake; modelovací nástroj; výpočetní pipeline; StencilJS; TypeScript; webová komponenta; webová aplikace

Abstract

This master's thesis aims to create a tool for visual modeling of computing Snakemake pipelines and to implement this tool into a remote pipeline management system. First of all, the reader is acquainted with this system and with the Snakemake tool. Then, existing tools solving the given issue are researched. In order to realize the solution, different approaches are analyzed, followed by designing a specific solution that proposes extending a web based tool named Elsa Designer. The solution is implemented according to this design and integrated into the pipeline management system. Automatic tests for the realized solution are created and their execution is documented and evaluated.

Keywords

Snakemake; modeling tool; computing pipeline; StencilJS; TypeScript; web component; web application

Poděkování

Nejprve bych rád poděkoval vedoucímu práce, Ing. Janu Kožusznikovi, Ph.D., za konzultace k práci a poskytnuté rady. Dále bych chtěl poděkovat rodině a přátelům, kteří mě za doby vytváření této práce podporovali.

Obsah

Seznam použitých zkratek a symbolů

API	–	Application programming interface
DAG	–	Directed acyclic graph
DSL	–	Domain-specific language
GUI	–	Graphical user interface
HEAppE	–	High-End Application Execution
HPC	–	High-performance computing
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
JSON	–	JavaScript Object Notation
KNIME	–	The Konstanz Information Miner
LINQ	–	Language-Integrated Query
MVC	–	Model–view–controller
PWA	–	Progressive web application
WHL	–	Web HPC Launcher
YAML	–	YAML Ain't Markup Language

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Problematika analýzy a zpracování dat se na automatizaci váže už celou řadu let. Za účelem vytvoření automatizovaného řešení problému analýzy dat se často používá velký počet nástrojů, jež mezi sebou vytvářejí různé závislosti na vzájemné vstupy a výstupy. Kombinace těchto nástrojů dává vzniknout tzv. *pipeline* (nebo také *workflow*), což je obvykle program nebo skript, který v definovaném pořadí spouští dílčí úlohy, předává jim vstupní data a jejich výstupy poté předává dalším úlohám opět jako vstupy.

Pro definici, údržbu a spouštění takovýchto pipelines existuje celá řada nástrojů. Tato diplomová práce navazuje na jeden z nich, a to sice Snakemake – škálovatelný pipeline nástroj ve formě nástavby programovacího jazyka Python. Má práce si klade za cíl vytvořit modelovací GUI nástroj pro vizuální definici Snakemake pipeline, který může uživatel využít k lepší organizaci kroků v pipeline, a tím docílit jednodušší tvorby a úpravy dané pipeline.

Dalším cílem této práce je rozšíření systému pro vzdálenou správu výpočetních pipeline o výše zmíněný modelovací nástroj. Tento webový systém má název Web HPC Launcher (WHL) a jeho rozšíření by mělo umožňovat definici Snakemake pipeline pomocí vizuálního nástroje zmíněného výše a také spouštění úloh dle takto specifikované pipeline.

K dosažení těchto cílů si ve své diplomové práci kladu dílčí cíle. Prvním z nich bude *state-of-the-art* rešerše nástrojů pro modelování výpočetních pipelines včetně přehledu jejich schopností, čímž se zabývá kapitola 2. V kapitole 3 bude následovat návrh vlastního přístupu a poté implementace řešení rozebraná kapitolou 4. Její součástí pak bude i popis toho, jak bylo řešení integrováno do systému WHL. Kapitola 5 se zaměří na testování řešení pomocí automatických testů v rámci definovaných testovacích scénářů a testovacích dat. Na závěr zhodnotím realizované řešení, popíši jeho omezení a další možný směr jeho rozvoje.

Kapitola 2

Seznámení s problematikou modelování výpočetních pipelines

Tato kapitola má za cíl seznámit čtenáře se systémem WHL a poskytnout náhled do problematiky tvorby výpočetních pipelines. První sekce se věnuje popisu systému WHL, na který tato práce navazuje. Následovat bude úvod do tvorby pipelines pomocí nástroje Snakemake. Ve zbývající části této kapitoly pak lze nalézt přehled používaných state-of-the-art nástrojů pro vizuální modelování pipelines v různých problémových doménách.

2.1 Web HPC Launcher

Systém Web HPC Launcher (WHL) je vyvíjená webová aplikace pro spouštění a správu výpočetních úloh na HPC clusterech superpočítačového centra IT4Innovations. Ve své finální podobě by měl uživatelům poskytovat možnosti vytváření, spouštění a správu úloh na daných výpočetních clusterech, včetně průběžného prohlížení výstupů jednotlivých úloh či nahrávání velkoobjemových vstupních dat.

WHL je webová aplikace typu ASP.NET Core MVC v kombinaci s technologiemi jazyků JavaScript a TypeScript. Ve své současné podobě obsahuje komponenty pro nahrávání velkoobjemových dat pomocí *chunked-file* nahrávání (nahrávání souborů po částech), základní autentizaci uživatelů, základní možnosti pro správu výpočetních úloh, která probíhá prostřednictvím HTTP komunikace s middlewarem HEAppE, a konfiguraci výpočetní Snakemake pipeline pomocí dynamického formuláře.

2.2 Snakemake

Snakemake [1] je nástroj sloužící k definici výpočetních pipelines prostřednictvím doménově specifického jazyka (DSL), tedy počítačového jazyka zaměřeného na doménu konkrétní problematiky. Toto

DSL nástroje Snakemake má formu rozšíření syntaxe jazyka Python o další koncepty, jež lze využít k popisu pipeline. Soubor s pipeline definovanou touto rozšířenou syntaxí se pak nazývá Snakefile.

2.2.1 Struktura Snakefile

Základní Snakefile se skládá z pravidel (*rules*), která jsou do jisté míry analogická k jednotlivým krokům v řetězci analýzy a zpracování dat. Každé pravidlo (až na specifické výjimky, viz níže) značí způsob, jakým vytvořit výstupní data z dat vstupních. Popis pravidla se tedy skládá z:

- názvu pravidla,
- definice vstupních a výstupních souborů,
- shell příkazu nebo Python kódu, jenž je zodpovědný za transformaci vstupních dat na data výstupní,
- popisu dalších aktivit spojených s vykonáváním daného pravidla, např. logování informací ohledně průběhu pravidla do souboru nebo výpis uživatelem definované zprávy s informacemi o pravidlu do konzole.

Výjimku v rámci této struktury může tvořit např. pravidlo, které obsahuje jen název a definici vstupních dat. Takovéto pravidlo pak slouží ke sbírání výsledných dat, jež mají být konečným výstupem celé pipeline (z toho důvodu takovýto typ pravidla nazývám *collector*).

Jednoduchý příklad Snakemake pipeline definované podle popsané struktury může vypadat třeba takto:

```
1  datasets = [ "dataset" + i for i in range(10) ]
2
3  rule all:
4      input: [ "mapped_reads" + ds + ".bam" for ds in datasets ]
5
6  rule bwa_map:
7      input:
8          "data/genome.fa",
9          "data/samples/{sample}.fastq"
10     output:
11         "mapped_reads/{sample}.bam"
12     shell:
13         "bwa mem {input} | samtools view -Sb - > {output}"
```

Listing 2.1: Ukázková Snakemake pipeline

Ukázkový kód 2.1 lze rozdělit do tří částí: blok s Python kódem, pravidlo `all` a pravidlo `bwa_map`. Při spuštění nástroje Snakemake nad tímto Snakefile se nejdříve vykoná blok s Python kódem. Pravidlo `all` představuje collector pravidlo. Pokud se nástroj Snakemake spustí bez argumentu specifikujícího požadovaná výstupní data, ze všech pravidel v definovaném Snakefile se ve výchozím stavu vyhodnocuje nejdříve první zapsané pravidlo ve Snakefile (v případě kódu 2.1 je to právě collector `all`).

Nástroj Snakemake poté rekurzivně vyhodnocuje závislosti pravidel podle vyhledávání odpovídajících definic vstupních a výstupních dat. Při vyhodnocování pravidla `all` tedy jako vstupní závislost nalezne výstup pravidla `bwa_map`, které opět vyhodnotí. Pro vstup tohoto pravidla již Snakemake nenajde v daném Snakefile žádné pravidlo, jež by tuto závislost splňovalo, proto bude vstupní data hledat v pracovním adresáři. Z těchto vstupních dat se pak vytvoří data výstupní aplikováním zadaného shell příkazu.

Snakefile syntaxe umožňuje použití zástupných proměnných zvaných *wildcards* ve formátu typu `{wildcard}`. Jejich použití lze v ukázce vidět na řádcích 9, 11 a 13 ve složených závorkách. *Wildcard* `{sample}` se vyhodnotí na části názvů všech souborů se souhlasným formátem zápisu, přičemž výsledné hodnoty této *wildcard* na řádcích 9 a 11 si vzájemně odpovídají. Wildcards `{input}` a `{output}` pak představují expanze názvů vstupních a výstupních dat (definovaných ve stejnojmenných direktivách daného pravidla) na jednotlivé hodnoty. Zadaný shell příkaz se tedy vykoná ve všech variacích odpovídajících vstupů a výstupů.

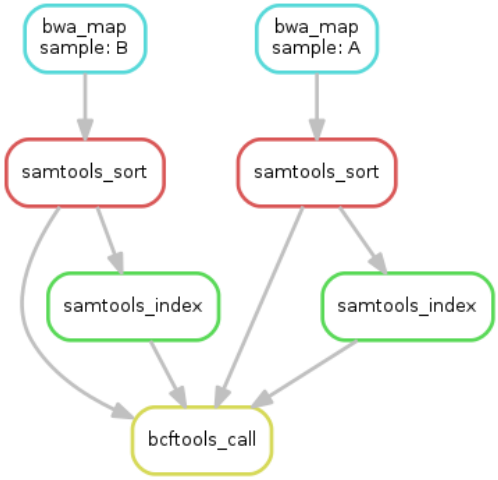
2.2.2 Snakemake engine

Plánování a vykonávání Snakefile pravidel (souhrnně nazýváno *Snakemake engine*) začíná tvorbou acyklického orientovaného grafu (DAG – např. obr. 2.1), který představuje plán pro vykonávání jednotlivých pravidel. Tyto výpočetní úlohy jsou v daném grafu reprezentovány uzly. Hrany mezi uzly představují výše zmiňované závislosti mezi pravidly (úlohami) – obecně hrana vedoucí z uzlu **A** do uzlu **B** značí, že úloha **B** potřebuje výstup úlohy **A** jako svůj vstup.

Každé dvě disjunktní cesty ve výsledném grafu představují řetěz úloh, jenž je možné vykonávat nezávisle na ostatních. Proto je nástroj Snakemake vhodný i k paralelizaci jak na jednom vícejádrovém stroji, tak i na výpočetních clusterech se sdíleným úložištěm. Vykonávání úloh pak lze škálovat bez nutnosti úprav samotné pipeline.

2.3 Přehled nástrojů pro modelování pipelines

Způsoby definic výpočetních pipelines lze zařadit do několika kategorií [3]. Do první kategorie spadají workflow systémy jako SciPipe nebo COMPSs, které pipelines definují použitím nějakého obecného programovacího jazyka (např. Python či Scala). Takovéto systémy typicky nepotřebují



Obrázek 2.1: Příklad DAG jako model závislostí mezi úlohami [2]

GUI, a jsou tedy vhodné pro nasazení v rámci serverových prostředí. Pro uživatele jejich využití ovšem předpokládá znalost daného programovacího jazyka.

Další kategorii tvoří např. systémy Nextflow nebo výše popsaný Snakemake, který je součástí problematiky této diplomové práce. Tyto systémy rozšiřují předchozí kategorií o použití jazyků DSL, jež slouží jako nástavba nějakého obecného programovacího jazyka (např. rozšíření jazyka Python o Snakefile syntaxi). Systémy této kategorie sdílí vlastnosti kategorie předchozí, navíc díky specializaci DSL na konkrétní doménu je vylepšená čitelnost kódu a jeho rozšířitelnost.

Systémy řadící se do třetí kategorie (např. Popper) definují pipelines použitím ryze deklarativního přístupu, kdy je využito zápisu pipeline do konfiguračního souboru (např. formátu YAML). Tento přístup dále zlepšuje čitelnost a použitelnost uživateli bez programovacích znalostí. Nevýhodou je ale ztráta možností imperativního a funkcionálního programování, čímž dochází k redukci víceúčelovosti a přizpůsobitelnosti výsledné pipeline.

V další kategorii se nacházejí systémově nezávislé jazyky (např. CWL nebo WDL) pro specifikaci pipelines. Pipelines definované těmito jazyky jsou následně parsovány a spouštěny nástroji třetích stran (Cromwell, Tibanna), které zajišťují běh pipeline na dané platformě. Oproti předchozí kategorii zde vzniká výhoda přenositelnosti pipeline na potenciálně jakoukoliv platformu. Další výhodou je to, že takovéto systémově nezávislé jazyky často podporují kompatibilitu s ostatními jazyky pro definici pipeline.

Do poslední kategorie se řadí systémy umožňující definici pipeline modelováním pomocí vizuálního nástroje. Jejich největší výhodou je jednodušší osvojení uživatelem a jistá míra abstrakce od konceptů vyžadujících programovací znalosti. Příklady takovýchto systémů jsou KNIME, Galaxy nebo Elsa Workflows. Vzhledem k tomu, že se tato diplomová práce zabývá problematikou v oblasti právě této kategorie, následuje podrobnější přehled těchto systémů.

2.3.1 KNIME

The Konstanz Information Miner (zkráceně KNIME) je prostředí pro vizuální sestavení a interaktivní spouštění datové pipeline [4]. V současnosti KNIME poskytuje dvě samostatné platformy: open source software KNIME Analytics Platform a komerční prostředí KNIME Server [5].

KNIME Analytics Platform obsahuje sadu nástrojů pro definici a vizuální modelování pipeline a její následné spouštění. Toto prostředí je implementováno v jazyce Java jako plugin pro vývojové prostředí Eclipse. KNIME Server poskytuje nástroje pro nasazení a automatizaci navržených pipelines, týmovou kolaboraci a management uživatelů a pipelines.

KNIME umožňuje extrakci, transformaci a analýzu dat pocházejících z různých datových zdrojů (např. Excel/CSV tabulek, databází nebo nestrukturovaných datasetů jako obrázky či dokumenty). Data lze zpracovat pomocí úkonů jako filtrování, seskupování, agregace, normalizace či vzorkování. KNIME poskytuje možnosti pro datovou analýzu a *data mining*, mezi nimiž lze nalézt např. shlukování, statistickou analýzu, tvorbu reportů a grafického znázornění dat.

V rámci datové analýzy KNIME nabízí i schopnosti strojového učení a umělé inteligence – lze budovat modely strojového učení mj. pro klasifikaci, redukci dimenze nebo regresi pomocí metod jako rozhodovací stromy, asociační pravidla či tzv. *deep learning*. Tyto modely lze následně optimalizovat, validovat a využít k tvorbě predikcí.

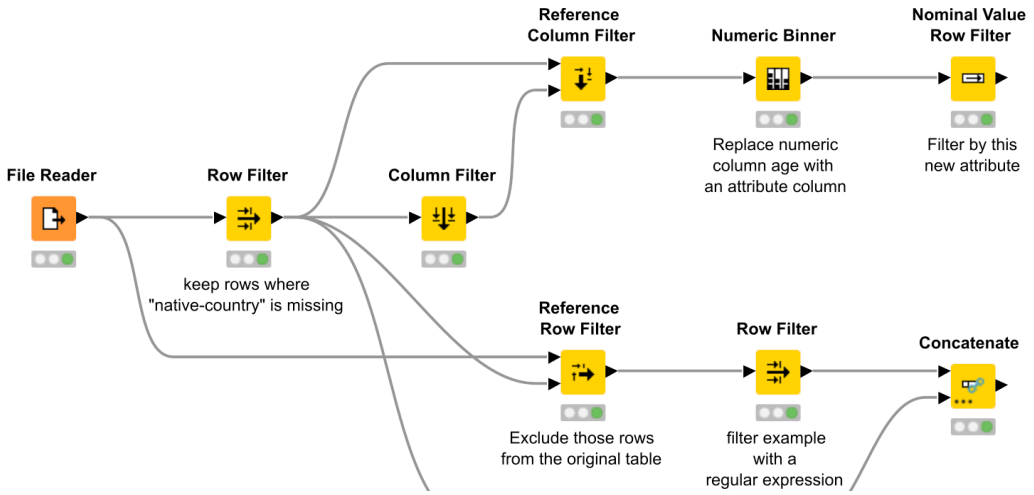
Samotná pipeline je v prostředí KNIME tvořena propojenými uzly, které reprezentují dané kroky datového zpracování a analýzy. Pro potřeby uživatelů poskytuje prostředí KNIME výchozí sadu uzlů, přičemž vzhledem k open source povaze softwaru KNIME Analytics Platform je možné použít uzly vytvořené komunitou nebo implementovat vlastní uzly. Jednotlivé uzly jsou pak konfigurovány GUI formuláři; do funkcionality uzlu lze zahrnout i uživatelem napsaný kód v Pythonu, R nebo JavaScriptu.

Mezi typické příklady použití KNIME patří analýza zákaznických dat, tvorba predikčních modelů pro chování zákazníků, tvorba statistických reportů a interaktivních grafických prezentací, analýza chování uživatelů na sociálních sítích či detekce anomálií.

2.3.2 Galaxy

Galaxy je webová platforma určená k datově intenzivnímu biomedicínskému výzkumu [6]. Tato platforma má formu webového portálu, prostřednictvím něhož mohou uživatelé nahrávat vstupní data, používat různé nástroje k provádění analýzy biomedicínských dat, modelovat, konfigurovat a spouštět pipelines a vizualizovat výsledky prováděných úloh. Důraz se zde dává na znovupoužitelnost navržených analytických řešení a možnost spouštět analýzy nad velkým množstvím velkoobjemových dat.

Svou webovou platformu Galaxy poskytuje v podobě open source softwarového frameworku, jenž může kdokoliv použít k nasazení a přizpůsobení vlastní Galaxy serverové instance na serverových strojích, výpočetních clusterech či cloudových službách [7]. Projekt Galaxy nabízí pro veřejné



Obrázek 2.2: Ukázková KNIME pipeline [5]

využívání svou primární instanci *usegalaxy.org* [8], přičemž k dispozici je dalších cca 130 veřejných instancí upravených pro účely specifických domén.

Uživatel má na platformě možnost používat nástroje pro zpracování a analýzu vstupních dat buď jednotlivě, nebo nástroje v grafickém návrhovém prostředí spojovat skrze jejich vstupy a výstupy do výpočetních pipelines. Konfigurace nástrojů pak probíhá prostřednictvím GUI formulářů.

Mezi nabízenými nástroji lze nalézt nástroje pro manipulaci se soubory genomů, genomickou analýzu, textovou manipulaci či biomedicínskou statistickou analýzu a vizualizaci. Podobně jako u KNIME mají uživatelé platformy Galaxy možnost využívat komunitně vyvíjené nástroje, případně vyvinout a publikovat vlastní nástroje.

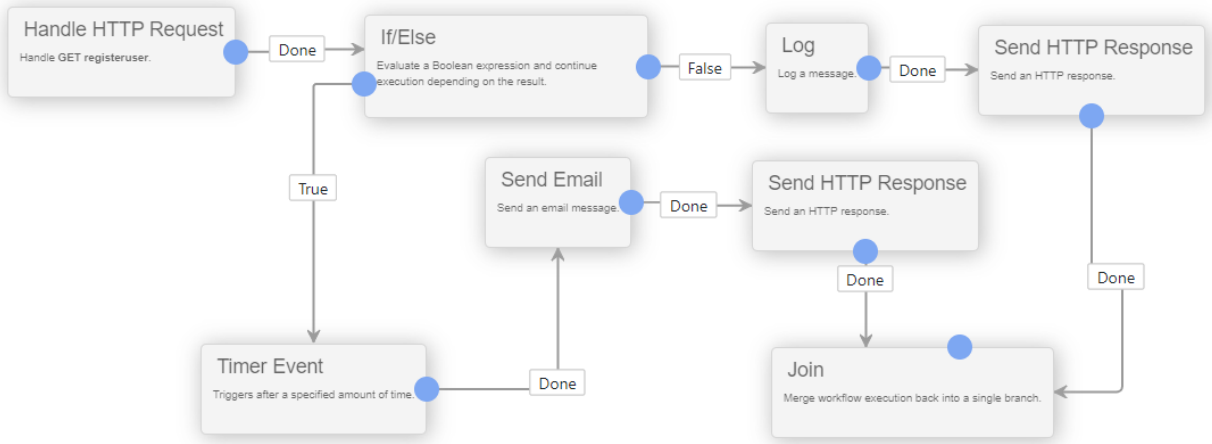
Platforma Galaxy je určena výzkumníkům bez programovacích nebo administrátorských znalostí. Ačkoliv byla tato platforma původně zaměřena především na výzkumnou analýzu v oblasti genomiky, v současnosti je Galaxy využívána ve vědeckých doménách nejen biologické povahy, ale např. i v oblastech obrazové analýzy, výpočetní chemie, lingvistiky nebo kosmologie.

2.3.3 Elsa

Elsa je knihovna a skupina nástrojů pro tvorbu a spouštění workflows v doméně aplikační business logiky [9]. Do projektu Elsa spadá knihovna pro .NET Standard, nástroj Elsa Dashboard pro správu workflows a modelovací nástroj Elsa Designer.

Knihovna Elsa umožňuje vytvářet a spouštět workflows zaměřené na business logiku pomocí sady programovacích API. Knihovnu lze zabudovat do jakéhokoliv .NET Standard projektu [10] (např. ASP.NET Core Blazor, MVC, Razor Pages). Kromě integrace do frontendové aplikace lze knihovnu Elsa využít i pro vybudování workflow služby, se kterou pak mohou externí aplikace interagovat.

Samotná definice workflow v rámci knihovny Elsa probíhá psaním C# kódu využívajícího API knihovny, což doplňuje možnost implementovat workflows i jako soubory formátu JSON/YAML.



Obrázek 2.3: Ukázková Elsa workflow

Jednotlivé úkony dané workflow jsou reprezentovány aktivitami. K dispozici je základní sada aktivit pro ovládání toku workflow, zpracování HTTP požadavků, logování nebo zasílání emailových zpráv. Knihovna pak také poskytuje možnost definice vlastních aktivit.

Elsa Dashboard je GUI nástroj, pomocí něhož jsou uživatelé schopni vizuálně definovat workflows a po jejich spuštění prohlížet výsledné stavy workflows i jednotlivých aktivit v dané workflow (tj. zda se aktivity spustila, případně jestli úspěšně či zda je v blokovaném stavu). Vizuální definice workflows je umožněna komponentou Elsa Designer, jež je součástí nástroje Dashboard. Nástroj lze přidat do jakéhokoliv ASP.NET Core projektu.

Nástroj Elsa Designer je k dispozici ve dvou podobách: jednak jako součást nástroje Elsa Dashboard a jednak jako stand-alone webová HTML5 komponenta. V Designeru je možné modelovat workflow propojováním bloků aktivit, kdy hrana vedoucí mezi aktivitami definuje pořadí vykonání aktivit, nedochází tedy k toku dat mezi aktivitami. Data lze pak mezi aktivitami sdílet pomocí aktivity pro deklaraci proměnné a využití výrazů Javascriptové nebo Liquid syntaxe pro následnou referenci na definované proměnné.

Aktivity lze v Elsa Designeru konfigurovat prostřednictvím formulářových oken. Kromě samotného modelování lze workflows importovat či exportovat do formátů JSON, YAML, XML, nebo v binární podobě. Nástroj Dashboard pak umožňuje i verzování workflows namodelovaných v Designeru.

Kapitola 3

Návrh vlastního řešení

V této kapitole se věnuji návrhu řešení problematiky vizuálního modelování Snakemake pipelines. Bude zde nejprve rozebrána volba obecného přístupu k návrhu řešení včetně volby typu realizované aplikace. V další sekci pak lze nalézt popis technologií použitých k realizaci řešení, jež se váží na konečný zvolený přístup. Popis bude zahrnovat základní charakteristiky těch nejdůležitějších z použitých technologií, proto se v rámci stručnosti a přehlednosti některé technologie vyskytnou až v následujících kapitolách.

Poslední sekce pak detailně popíše návrh řešení, podle něhož bylo řešení následně realizováno. Bude zde kladen důraz na návrh uživatelského prostředí jeho mechanismů spolu s návrhem integrace řešení do systému WHL. Části návrhu pak budou doplněny názornými diagramy.

3.1 Analýza přístupu k návrhu a realizaci řešení

V rámci volby přístupu za účelem vytvoření modelovacího nástroje pro Snakemake pipelines je možno volit mezi dvěma základními přístupy. Jedna z možností je použití a přizpůsobení již existujícího řešení. Vzhledem k tomu, že neexistuje žádný nástroj přímo uzpůsobený pro účely vizuálního modelování Snakemake pipelines, nelze k návrhu a implementaci řešení do systému WHL žádný takový nástroj využít.

Případně lze zvolit již existující modelovací nástroj určený k modelování pipelines odlišných od nástroje Snakemake, přičemž by bylo třeba následně navrhnout způsob konverze pipelines vytvořených daným nástrojem na pipelines pro nástroj Snakemake. Takový přístup má nevýhodu v tom, že formát, do kterého jsou pipelines daného modelovacího nástroje ukládány či exportovány, nemusí nutně zůstat v průběhu času a vývoje daného nástroje stejný. Tento přístup by tak umožňoval pouze konverzi v rámci jedné nebo několika verzí modelovacího nástroje a v případě změn ve formátu vstupní pipeline by bylo nutné aktualizovat také konverzní nástroj.

Druhou možností se pak nabízí vytvoření modelovacího nástroje pro Snakemake pipelines od základů. Kromě nevýhody v podobě zjevně větší časové náročnosti na realizaci řešení je tu ovšem

i hlavní výhoda v možnosti lepšího přizpůsobení modelovacího nástroje potřebám Snakemake. Lze tak zvolit nejvhodnější typ aplikace a s tím i způsob nasazení tohoto modelovacího nástroje do systému WHL.

3.1.1 Volba typu aplikace

Co se typu aplikace týče, zmiňovaný nástroj pro modelování pipelines lze realizovat buď jako desktopovou aplikaci, nebo jako webovou aplikaci. Systém WHL je webová aplikace, tudíž uživatelská interakce se systémem probíhá prostřednictvím webového prohlížeče. Z tohoto důvodu by se modelovací nástroj měl co nejlépe integrovat do tohoto webového prostředí tak, aby nástroj co nejlépe utilizoval toho prostředí a aby práce s nástrojem byla pro uživatele co nejpohodlnější.

U desktopové aplikace by integrace do systému WHL probíhala vzdálenou HTTP komunikací pomocí definovaného API. Tato komunikace by také probíhala v případě webové alternativy, zde ovšem vzniká výhoda v podobě možnosti integrace modelovacího nástroje jako webové komponenty, kdežto u desktopového řešení by se jednalo o samostatnou aplikaci. V případě webového řešení by u komunikace skrz API odpadla nutnost implementovat další autentizační a autorizační mechanismus díky již existujícímu mechanismu v rámci ASP.NET Core.

Webová aplikace obecně může navíc využít výhody odlehčené aktualizace samotné aplikace. V případě nové verze modelovacího nástroje by stačilo znovu načíst webovou stránku s nástrojem – toto ovšem předpokládá relativně malý *application footprint*, tedy především malou velikost aplikace, aby načítání netrvalo moc dlouho a nespotřebovalo velké množství datového přenosu. Pro umožnění aktualizace desktopové varianty by musel vzniknout mechanismus, jenž by tuto funkcionalitu zajišťoval. Především kvůli webové povaze systému WHL, ale také kvůli ostatním výše zmíněným výhodám, jsem se rozhodl modelovací nástroj realizovat také jako webovou aplikaci.

3.1.2 Výběr přístupu

Na začátku této sekce jsem rozebíral potenciální výhody a nevýhody dvou základních přístupů k řešení problematiky této práce: využití existujícího nástroje nebo tvorba od základů. Ještě je k dispozici jedna možnost, jež kombinuje obě předchozí, a to sice možnost zvolit nějaký existující modelovací nástroj, který již poskytuje sadu základních funkcionalit, jež by byly použitelné i v mém řešení.

Tímto způsobem by bylo možné buďto vybrat z původního nástroje části vhodné i v rámci mého řešení, nebo rozšířit či přetvořit původní nástroj k účelům této práce. K využití existujícího nástroje by tak nedocházelo kvůli záměru konverze pipelines daného nástroje na pipelines pro nástroj Snakemake, nýbrž kvůli využití již implementované základní funkcionality jako vykreslování vizuálních bloků, spojování bloků hranami, vykreslování modálních oken apod.

V podsekci 2.3.3 se nachází zmínka o tom, že modelovací nástroj Elsa Designer existuje i jako webová komponenta. Tato komponenta je dostupná v open source podobě pod licencí MIT, tudíž

se nabízí ji využít v rámci výše popsaného kombinovaného přístupu. Komponenta v rámci prohlížeče implementuje uživatelské rozhraní pro přidávání bloků, jejich propojování a konfiguraci skrz formuláře, a ukládání stavů bloků a hran mezi nimi. Právě tyto prvky jsou použitelné i v realizaci mého řešení, proto jsem jako přístup k řešení problematiky této práce zvolil způsob přizpůsobení komponenty Elsa Designer účelům pipelines nástroje Snakemake.

3.2 Použité technologie

Obsah této sekce tvoří výčet technologií, jež budou využity v rámci následné realizace řešení. Tyto technologie mají především webovou povahu a zahrnují jak samotné programovací jazyky, tak i platformy a frameworky na nich vybudované.

3.2.1 HTML

HyperText Markup Language (HTML) je značkovací jazyk určený k definici struktury obsahu na webových stránkách [11]. Spolu s CSS a JavaScriptem se jedná o základní stavební blok webových stránek a aplikací. HTML je pouze značkovací jazyk, tudíž nedisponuje vlastnostmi programovacích jazyků k tvorbě dynamických programů. Samotná definice obsahu v HTML probíhá členěním obsahu do předdefinovaných elementů, jež mohou zahrnovat i informace obsažené v atributech. Ty pak dále specifikují způsob členění obsahu a mohou se vázat i na použití dalších webových technologií (CSS, JavaScript).

Specifikace jazyka HTML byla v minulosti spravována dvěma odlišnými entitami: konsorciem W3C a komunitou WHATWG [12]. V současnosti tyto dvě entity společně spravují konsolidovanou verzi specifikace HTML s názvem HTML Living Standard. Specifikace definuje mj. syntaxi jazyka, elementy a způsoby jejich užití či přístupy k uživatelské interakci.

3.2.2 CSS

Zatímco HTML poskytuje možnosti pro stavbu struktury webové stránky či aplikace, možnosti tohoto jazyka v rámci vizuální úpravy a členění definovaného obsahu jsou v tomto jazyce obsaženy jen v základním rozsahu, a to především ve formě uspořádání obsahu podle logického sledu informací. Tento nedostatek odstraňuje jazyk Cascading Style Sheets (CSS) určený k popisu vizuální prezentace HTML dokumentů [13].

Pro popis vizuální reprezentace (stylování) obsahu jsou v CSS k dispozici pravidla. Součástí každého pravidla je selektor definující kontext použití daného pravidla a jeden nebo více stylových atributů, jež ovlivňují vzhled elementu v rámci daného selektoru. Kromě specifických vizuálních úprav vlastností jako velikost písma, barva či odsazení obsahu CSS nabízí např. i možnosti flexibilních úprav členění a řazení obsahu pomocí *flexbox layoutu* či tvorby základních animací.

Při tvorbě projektů rozsáhlejšího charakteru se mohou CSS soubory stát nepřehlednými a hůře upravitelnými. Navíc CSS, podobně jako HTML, nemá možnosti programovacích jazyků, které by bylo vhodné využít např. k automatickému vygenerování stylů či znovupoužití stylů ve vnořených stylech. Za tímto účelem lze využít tzv. CSS preprocesor, jenž tvoří nástavbu CSS syntaxe a obohacuje CSS o další mechanismy [14]. Konsorcium W3C spravuje specifikaci jazyka CSS – místo jedné ucelené specifikace však pravidelně vzniká tzv. *snapshot* všech dílčích specifikací jazyka CSS.

Mezi nejpoužívanější CSS preprocesory patří Sass a LESS. Oba tyto preprocesory poskytují vzájemně podobné možnosti, mezi něž patří např. tvorba vnořených pravidel, deklarace a využívání proměnných, cykly ke generování stylů či znovupoužívání stylů pomocí tzv. *mixins*. Webová komponenta Elsa Designer ke stylování používá preprocesor Sass, proto jej budu používat také.

3.2.3 JavaScript

JavaScript je programovací jazyk, jenž byl za dobu své existence používán především k poskytnutí možnosti spouštět skripty webových stránek na straně uživatele ve webových prohlížečích [15]. V posledních letech se ovšem rozšířil i mimo webový prohlížeč a své využití najde i v prostředích jako Node.js, React Native nebo Electron.

Jedná se o objektově orientovaný programovací jazyk, jenž se v prostředí webového prohlížeče používá k tvorbě stránek s dynamickým obsahem a chováním. Lze tak pomocí něj mj. upravit výchozí chování elementů na stránce, zasílat a přijímat asynchronní zprávy či rozšířit možnosti uživatelské interakce. JavaScript je spravován specifikací ECMAScript pod organizací Ecma International. Všechny moderní webové prohlížeče a většina serverových prostředí splňuje standard ECMAScript 5.1 z roku 2011, přičemž nejnovější verzí standardu je ECMAScript 2020.

3.2.4 TypeScript

JavaScript provádí typové kontroly objektů dynamicky až za běhu prostřednictvím tzv. *duck typingu*, kdy typová vhodnost objektu pro nějaké specifické využití je ověřena na základě toho, zda objekt obsahuje atributy či metody nutné pro dané využití [16]. Tento přístup typové kontroly může do kódu uvést chyby, jež se mohou následně projevit až při jeho spuštění, tudíž jim nelze předejít v rámci statické kontroly kódu.

Za účelem statické typové kontroly JavaScriptového kódu vznikl jazyk TypeScript [17]. Tento jazyk z hlediska syntaxe a sémantiky představuje nadmnožinu JavaScriptu, jenž rozšiřuje o statické typové definice. Veškerý validní JavaScriptový kód je tedy, až na potenciální chyby typové kontroly, zároveň i validním TypeScriptovým kódem.

Kód napsaný v TypeScriptu je pro výsledné použití v prohlížečích či aplikacích transformován do JavaScriptu pomocí jednoho z kompilátorů, jež lze integrovat v rámci vývojových nástrojů či *runtime* prostředí jako Node.js. V rámci webových aplikací je tak možno docílit překladu kódu na vyžádání za účelem jednoduššího a automatizovaného nasazení kódu, nebo lze kód přeložit ještě

před spuštěním aplikace a pak jej nasadit jako běžný JavaScript např. v prostředích, jež možnost dynamického nasazení kódu nepodporují.

Specifikace jazyka TypeScript je spravována společností Microsoft. Poslední verzí této specifikace je TypeScript 4.1 z roku 2020. Jazyk kromě typové kontroly přidává do JavaScriptu i další vlastnosti jako rozhraní, jmenné prostory či výčtové typy.

3.2.5 Node.js

Node.js je serverové runtime prostředí schopné spouštět JavaScriptový kód mimo prohlížeč [18]. Kvůli své událostmi řízené architektuře je toto prostředí vhodné zejména k tvorbě serverových aplikací a dlouhodobě běžících procesů. Node.js je nízkoúrovňová platforma, na které je vystavěna celá řada technologií a frameworků jako Express, Meteor, Socket.io či Angular. Node.js je distribuovaně vyvíjeno jakožto open source platforma organizací OpenJS Foundation. Nejnovější hlavní verzí je Node.js 15.

Prostředí Node.js je navrženo v rámci událostmi řízené architektury, tudíž ke zpracování požadavků nepoužívá vlákna, nýbrž mechanismy pro vyvolávání a obsluhu událostí. Asynchronní přístup těchto mechanismů umožňuje efektivnější využití výpočetních zdrojů, přičemž Node.js umožňuje i paralelní chod programů prostřednictvím optimalizované komunikace mezi vícero procesy.

3.2.6 StencilJS

Za účelem efektivnějšího vývoje a překladu webových aplikací a jejich komponent lze využít specifické frameworky či nástroje. Jedním z těchto nástrojů je StencilJS [19]. Jedná se o kompilační nástroj původně vytvořený pro účely frameworku Ionic, v současné podobě je však využitelný i samostatně.

StencilJS je zaměřen na sestavování webových komponent. Webová komponenta je sada technologií, jež umožňuje tvorbu vlastních znovupoužitelných elementů [20]. Tyto elementy v sobě skrývají funkcionalitu zapouzdřenou oproti zbytku webové aplikace; je tak zajištěno to, že kód komponenty nebude kolidovat s vnějším kódem aplikace.

Webové komponenty se skládají z třech dílčích technologií. První z nich jsou vlastní elementy (anglicky *custom elements*), jež představují sadu JavaScriptových API pro definování vlastních elementů se specifickým chováním. **Shadow DOM** je druhá z těchto technologií taktéž ve formě JavaScriptových API a slouží k připojení vlastního DOM stromu k definovanému elementu, čímž jsou vnitřní vlastnosti a struktura komponenty izolovány od ostatních částí aplikace. Poslední technologii webových komponent tvoří HTML šablony ve formě elementů, jež lze využít k tvorbě znovupoužitelné HTML struktury.

StencilJS poskytuje sadu nástrojů k tvorbě a sestavení webových komponent. Takto sestavené komponenty pak lze integrovat buď do webové aplikace bez frameworku, nebo do aplikace vytvořené frameworkem pomocí vazby na daný framework (anglicky *framework binding*) za předpokladu, že

je daný framework nástrojem StencilJS podporován. Výše zmiňovaná komponenta Elsa Designer je sestavena bez vazeb na jakýkoliv framework, je tedy použitelná ve webové aplikaci bez frameworku.

Webové komponenty vytvořené v nástroji StencilJS podporují všechny moderní webové prohlížeče, a to buď nativně, nebo za použití tzv. *polyfills*, tedy kódu zajišťujícího specifickou funkcionalitu na prohlížeči, jenž tuto funkcionalitu nepodporuje nativně [21]. Tvorba webových komponent v nástroji StencilJS pak probíhá prostřednictvím jazyka TypeScript. K budování vnitřní HTML struktury komponenty je použit šablonovací nástroj JSX.

Kromě toho, že StencilJS poskytuje možnosti sestavení webových komponent za účelem využití v externích aplikacích, StencilJS poskytuje i vlastní runtime prostředí pro vývoj webových komponent či jejich nasazení. Toto runtime prostředí je pak nástavbou prostředí Node.js.

3.2.7 JSX

Šablonovacím nástrojem (anglicky *template engine*) používaným v rámci nástroje StencilJS je nástroj JSX. Šablony napsané v JSX umožňují kombinovat HTML strukturu a JavaScriptový kód bez nutnosti použití řetězců pro definici elementů, atributů a naslouchačů událostí.

Kromě obvyklých možností běžného template enginu (jako interpolace proměnných, podmíněné vykreslování či cykly) lze mezi schopnostmi nástroje JSX nalézt i možnosti mj. pro dynamické nastavení naslouchačů událostí v elementech, nastavení referencí na elementy či dynamické vkládání HTML obsahu do struktury komponent.

3.2.8 jsPlumb

Součástí webové komponenty Elsa Designer je i knihovna **jsPlumb**, která v rámci komponenty zajišťuje samotnou modelovací část nástroje, tedy vykreslování bloků a jejich spojování hranami. Tato knihovna tedy slouží k poskytnutí možnosti vizuálně spojovat elementy na webové stránce prostřednictvím SVG [22].

Knihovna jsPlumb je k dispozici ve dvou verzích: Community Edition (open source verze) a Toolkit Edition (komerční nástavba). Mezi dostupnými nastaveními knihovny jsPlumb lze nalézt např. *drag-and-drop* funkcionalitu, *callback* funkce na události jako vytvoření nebo odebrání hrany či tvorba základních animací.

3.2.9 C#

Za účelem integrace realizovaného řešení do systému WHL bude potřeba do tohoto systému implementovat funkcionalitu, jež tuto integraci zajistí. Systém WHL je založen na platformě ASP.Net Core, jejímž klíčovým programovacím jazykem je především jazyk C# [23].

C# je objektově orientovaný programovací jazyk sloužící k tvorbě široké škály aplikací v rámci ekosystému platformy .NET. Programy v jazyce C# jsou spouštěny na virtuálním běhovém systému

common language runtime (CLR) ve formě kompilovaného mezikódu. CLR pak mj. zajišťuje správu paměti pomocí tzv. *garbage collection* či správu výpočetních zdrojů.

Mezi hlavní vlastnosti a schopnosti jazyka C# patří mj. dotazovací jazyk LINQ, možnost spouštění asynchronních operací či přenositelnost kódu na různé typy platforem. Jazyk je vyvíjen společností Microsoft; poslední vydanou verzí jazyka je C# 9 podporovaná platformou .NET 5.

3.2.10 ASP.NET Core

Platforma .NET podporuje celou řadu frameworků, pomocí nichž lze vyvíjet aplikace za použití jazyku C# i dalších programovacích jazyků. Jedním z těchto frameworků je i ASP.NET Core [24], v němž je vyvíjen i systém WHL. Tento framework se používá k tvorbě *server-side* aplikací, přičemž pro vybudování uživatelského rozhraní podporuje i zahrnutí *client-side* frameworků jako Blazor, Angular či React.

Aplikace vyvíjené ve frameworku ASP.NET Core jsou spustitelné na platformě .NET Core – otevřená *cross-platform* implementace platformy .NET sloužící jako následník starší platformy .NET Framework. Systém WHL je vybudován na verzi frameworku .NET Core 3.1, jež je z hlediska jmenné konvence zároveň i verzí finální; nástupnická verze z roku 2020 již nese označení .NET 5.

3.3 Návrh aplikace

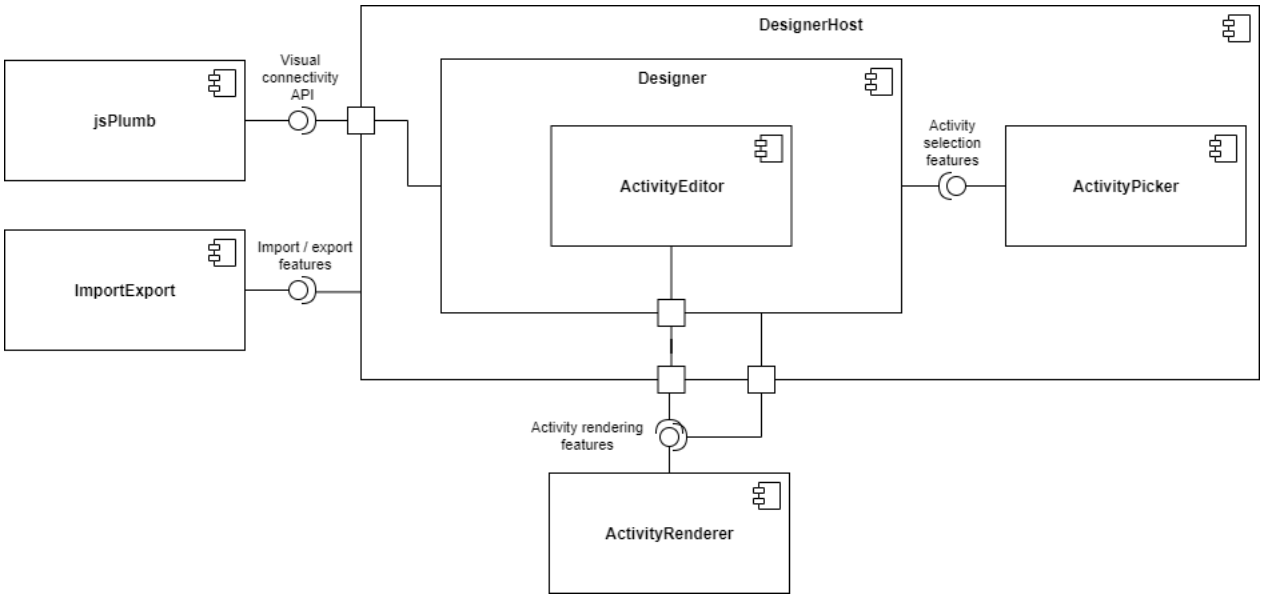
Tato sekce obsahuje detailnější popis návrhu aplikace, jež bude řešit problematiku této práce. Je zde zahrnut jak popis aplikace Elsa Designer, jež bude použita k realizaci řešení, tak i popis návrhu řešení včetně základních mechanismů aplikace a integrace do systému WHL.

3.3.1 Elsa Designer

Struktura aplikace Elsa Designer je tvořena několika základními komponentami. Komponenta **DesignerHost** registruje naslouchače událostí pro přidávání aktivit, jejich úpravu a aktualizaci a vyvolává import a export workflow. Součástí této komponenty je komponenta **Designer**, jež obstarává velkou část obecné funkcionality samotného modelovacího nástroje. Jedná se především o inicializaci workflow, nastavení knihovny jsPlumb a propojení jejích naslouchačů událostí s vysíláním událostí do jiných částí aplikace.

Komponenta **ActivityEditor** především definuje strukturu pro úpravu aktivity, tedy zobrazení modálního okna s formulářem k vyplnění dat aktivity. Komponenty Designer i ActivityEditor využívají komponentu **ActivityRenderer**, jež je zodpovědná za vykreslování aktivit ve dvou různých režimech (návrhovém a editačním) v závislosti na tom, zda se aktivita vykresluje na modelovací ploše či uvnitř editačního modálního okna.

Nabídku pro přidání nové aktivity na modelovací plochu obstarává komponenta **ActivityPicker**. Součástí nabídky je členění aktivit v rámci kategorií a možnost filtrování aktivit dle zadaného



Obrázek 3.1: Základní struktura aplikace Elsa Designer

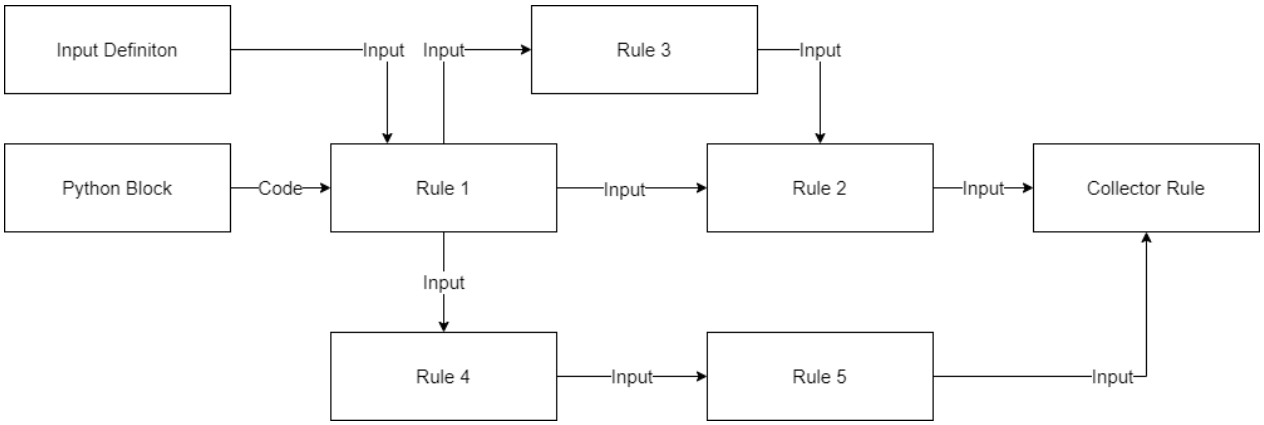
textu. Komponenta **ImportExport** pak zajišťuje konverzi a načítání workflows z různých souborových formátů (ve výchozím stavu formáty YAML, XML a JSON). Tuto základní strukturu aplikace Elsa Designer shrnuje diagram na obr. 3.1.

3.3.2 Návrh uživatelského rozhraní

Při návrhu modelovacího nástroje pro Snakemake pipelines je třeba brát v úvahu pár omezení. Prvním z nich je fakt, že soubory Snakefile jsou psány v kódu ve formě nástavby na syntaxi jazyka Python. Pokud by tak měl navržený nástroj co nejpřesněji modelovat Snakemake pipelines, prakticky by to znamenalo modelování kompletní Python syntaxe. Vytvoření takového nástroje však značně přesahuje problematiku této diplomové práce.

Druhé omezení představuje skutečnost, že i s vizuální pomocí modelovacího nástroje bude uživatel muset mít technické znalosti požadované k tvorbě Snakemake pipelines. Z těchto dvou omezení vyplývá kompromis v podobě vytvoření modelovacího nástroje, jenž místo kompletního modelování Snakemake syntaxe (a potažmo syntaxe Pythonu) bude spíše poskytovat možnosti modelování jakožto způsob grafického uspořádání Snakemake pravidel a vstupů a jejich základní propojení. Detailní nastavení pravidel a vstupů pak bude modelovací nástroj zajišťovat skrze dynamické interaktivní formuláře. Hlavní výhodu tak nástroj uživateli poskytne v přehlednější tvorbě pipelines při zachování modularity v podobě Pythonovského kódu.

Nástroj tedy bude po vzoru aplikace Elsa Designer poskytovat uživatelské rozhraní ve dvou módech: v návrhovém a v editačním. Návrhový mód bude představovat samotnou modelovací plochu, na níž bude probíhat přidávání Snakemake pravidel a dalších bloků a jejich spojování hranami. Jako



Obrázek 3.2: Ukázkový diagram návrhového módu navrhovaného modelovacího nástroje

další bloky jsem se rozhodl ztvárnit i blok vstupu, blok s Python kódem a blok pro speciální collector pravidlo.

Blok vstupu bude sloužit k definici vstupních dat. Místo toho, aby byla definice vstupních dat pouze součástí nastavení pravidla, zvolil jsem vstupní data navrhnout jako samostatný blok z důvodu znovupoužitelnosti skrz více pravidel. Python blok bude uživateli poskytovat možnost vložit Python kód do výsledného Snakefile, a to za účelem jakéhokoliv prvotního nastavení běhu nástroje Snakemake, jež bude uživatel vyžadovat.

Blok s collector pravidlem bude po vzoru mnou pojmenovaného collector pravidla (viz podsekce 2.2.1) představovat nepovinné zjednodušené vstupní pravidlo obsahující pouze definici vstupních dat. V případě, že bude toto pravidlo v namodelované pipeline přítomné, propíše se do výsledného Snakefile hned jako první pravidlo (po propsání případného Python kódu).

Sémantika hran v namodelovaném diagramu pak bude taková, že orientovaná hrana vedoucí ze vstupního bloku nebo bloku pravidla do bloku jiného pravidla bude značit datovou závislost cílového pravidla na výstupu zdrojového bloku (tedy bloku pravidla nebo bloku vstupních dat). Hrana vedoucí z Python bloku do jakéhokoliv jiného bloku bude mít spíše symbolický význam přítomnosti Python kódu; ten se vždy propíše na začátek výsledného Snakefile. Výše popsanou podobu návrhového módu znázorňuje zjednodušený ukázkový diagram na obr. 3.2.

Podoba editačního módu bude záviset na konkrétním typu bloku. V případě vstupního bloku je pouze třeba definovat vstupní data formou Python či Snakemake výrazu a případně ještě označit daný blok identifikátorem pro lepší přehlednost. Blok s Python kódem pak bude podobně obsahovat jen pole pro zadání Python kódu – zde ani nebude identifikátor potřeba kvůli tomu, že Python blok budeme v modelu pipeline očekávat nejvýše jednou.

Nejvíce možností nastavení bude poskytovat blok Snakemake pravidla. Ten kromě názvu pravidla bude obsahovat i definice svých vstupů a výstupů a také definice různých příkazů a činností, jež se mají vykonat při spuštění daného pravidla. Zde přijde na řadu interaktivita uživatelského rozhraní – pro definici vstupů pravidla jsem jako nejintuitivnější přístup zvolil takový, kdy vstupy budou

ztvárněny nabídkou se zaškrtávacími poli. Do této nabídky se propíší dostupné možnosti podle toho, jaké vazby vedou do daného pravidla z jiných pravidel či bloků vstupních dat. Uživatel pak bude volit takové vstupy, které jsou daným pravidlem vyžadovány.

V rámci definice výstupů pravidla bude mít uživatel možnost dynamicky přidávat výstupy pravidla prostřednictvím textového pole. Výstupy pravidla budou mít opět podobu Python/Snakemake výrazů a uživatel je po přidání bude moci zase odebrat z dynamického seznamu. Pravidlo pak bude poskytovat ještě pole pro vyplnění *run* a *shell* příkazů, nastavení pro logování a případně další nastavení, jež se Snakemake pravidla týkají. Model podoby editačního módu bloku s pravidlem lze nalézt na obr. 3.3. Blok s collector pravidlem bude skýtat pouze nastavení názvu pravidla a také definici vstupů pravidla, jež byla již popsána výše.

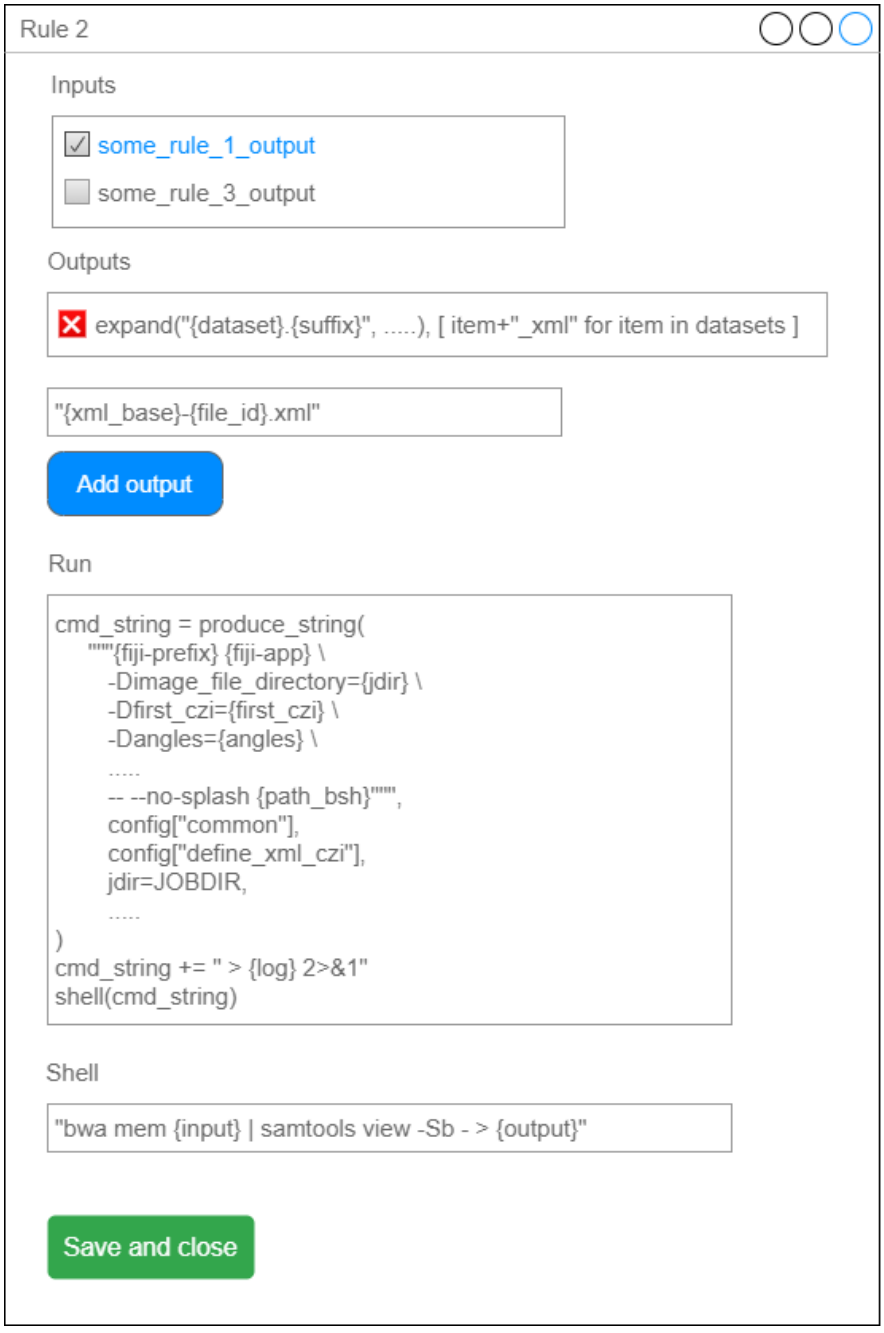
3.3.3 Návrh integrace do systému WHL

Vedle návrhu samotné aplikace pro modelování Snakemake pipelines je potřeba se zamyslet i nad tím, jakým způsobem bude tento modelovací nástroj fungovat v rámci systému WHL. Součástí zadání této diplomové práce je i implementace tohoto modelovacího nástroje do systému WHL tak, aby systém umožnil běh výpočetních úloh dle nástrojem namodelované pipeline. Integrace modelovacího nástroje v rámci systému WHL tak bude fungovat na základě komunikace nástroje s definovaným API na straně serveru systému WHL. Server pak bude zajišťovat tyto úkony:

- ukládání modelovaných pipelines v nástroji na serverové úložiště za účelem zálohování a exportu,
- načítání namodelovaných pipelines ze serveru zpět do modelovacího nástroje tak, aby uživatel mohl pokračovat v práci na pipeline,
- poskytnutí výčtu pipelines uživatele uložených na serveru,
- spouštění uživatelské pipeline uložené na serveru skrze vzdálený middleware HEAppE.

Tyto úkony bude server zajišťovat v kombinaci s mechanismy autorizace a autentizace uživatele. Uživatel bude mít přístup pouze ke svým pipelines, systém WHL tedy bude pipelines spravovat v rámci uživatelských účtů. Uživatel bude mít možnost si buďto vybrat správu pipelines skrz server díky funkcím serverového ukládání a načítání, nebo využít funkcionality importu a exportu pipelines za účelem jejich správy na svém lokálním úložišti.

Co se funkcionality spouštění pipelines týče, je zde třeba poznamenat, že v době tvorby této diplomové práce není možné implementovat tuto funkcionalitu pro úlohy spouštěné skrze middleware HEAppE, s nímž za účelem správy úloh systém WHL komunikuje. Typy spouštěných úloh a jejich nastavení totiž ve své podstatě závisí na tom, jestli middleware HEAppE poskytuje a implementuje specifické šablony pro běh daných typů úloh. Skrze middleware HEAppE tak nelze spouštět libovolné úlohy, nýbrž pouze úlohy, jež jsou tímto middlewarem podporované prostřednictvím daných šablon.



Obrázek 3.3: Ukázkový model editačního módu modelovacího nástroje

V době tvorby této práce ovšem middleware HEAppE neposkytuje šablonu pro spouštění libovolné Snakemake pipeline.

Ačkoliv nelze tuto funkcionalitu realizovat v rámci komunikace s middlewarem HEAppE, pořád je možné rozšířit systém WHL o prototyp této funkcionality tak, aby bylo demonstrováno správné chování této funkcionality na straně backendu systému WHL. Rozhodl jsem se tedy takový prototyp vytvořit, a to sice za použití mock-up verze middlewaru HEAppE, která bude také součástí implementační části této diplomové práce.

Vzdálená API tohoto mock-upu bude poskytovat předpřipravené odpovědi na HTTP požadavky přicházející ze strany backendu systému WHL dle specifikace API middlewaru HEAppE. Prototyp funkcionality spouštění pipelines pak bude v systému WHL připraven pokud možno co nejpřesněji také podle této specifikace do takové podoby, aby bylo možné jej rozšířit a použít i v kombinaci s API middlewaru HEAppE po úpravách na straně middlewaru HEAppE a drobných úpravách na straně backendu systému WHL.

Kapitola 4

Realizace řešení

Obsahem této kapitoly je popis implementace řešení v podobě, v jaké bylo představeno v předchozí návrhové kapitole. Tato kapitola je rozdělena na dvě části. V první části čtenář nalezne implementační detaily samotného modelovacího nástroje jakožto webové komponenty. Budou zde popsány dílčí aspekty implementovaného nástroje a rozebrána jeho funkcionalita z technického hlediska.

Druhá část kapitoly představí výsledné uživatelské rozhraní modelovacího nástroje, popíše jeho ovládání z uživatelského hlediska a jeho dílčí grafické prvky. Další část kapitoly se pak zabývá implementací integrace realizované komponenty do systému WHL. V této části bude popsáno, jak bylo nutné systém WHL za tímto účelem upravit, jak spolu komponenta a systém WHL komunikují a jakým způsobem probíhá nasazení komponenty do systému. V závěrečné části kapitoly poté bude popsán prototyp funkcionality spouštění namodelovaných pipelines.

4.1 Implementace modelovacího nástroje

K implementaci samotného modelovacího nástroje došlo za použití webové komponenty Elsa Designer a jejích zdrojových kódů. Realizované řešení z původní komponenty Elsa Designer přebírá především základní prvky uživatelského rozhraní. Bylo třeba tedy doimplementovat veškerou funkcionalitu spjatou s tvorbou Snakemake pipelines.

Detailním popisem těchto implementovaných záležitostí se zabývá právě tato sekce. Je nutno poznamenat, že implementované části nástroje zajišťující integraci nástroje včetně obsluhy načítání a ukládání pipelines na server (a tedy i komunikaci se serverem) budou popsány až v závěrečné části této kapitoly zabývající se integrací modelovacího nástroje do systému WHL.

4.1.1 Komponenty ve StencilJS

Webová komponenta či aplikace vytvořená ve StencilJS se skládá z dílčích jednotek zvaných komponenty. Komponenta ve StencilJS představuje samostatný zapouzdřený celek definující HTML strukturu jakožto znovupoužitelnou součást uživatelského rozhraní. Kromě samotné struktury lze

do komponenty přidat TypeScriptovou funkcionalitu ve formě metod; tyto metody pak rozšiřují komponentu o logiku, jež upravuje dynamické chování komponenty.

Struktura komponenty je definována v rámci návratové hodnoty metody **render** pomocí syntaxe JSX. Každá komponenta určuje název HTML elementu, který bude tuto strukturu reprezentovat v modelu DOM. Komponentu pak lze v podobě tohoto elementu využívat ve struktuře jiných komponent v aplikaci a také s ní zacházet jako s běžnou součástí DOM modelu (lze tedy např. používat její selektor v knihovně jQuery).

StencilJS pro účely nastavení atributů komponent a komunikace mezi komponentami nabízí využití několika dekorátorů. Tyto dekorátory rozšiřují způsoby, jakými mohou komponenty využívat jiných komponent v rámci aplikace. Dekorátor **@Prop** označuje takové atributy komponenty, jež jsou nastavitelné vnější (např. hostitelskou) komponentou. Lze zde nastavit mj. název HTML atributu, kterým se atribut vyznačuje v rámci odpovídajícího elementu v modelu DOM.

Dekorátor **@Method** podobně jako dekorátor @Prop také vystavuje části komponenty navenek; u tohoto dekorátoru se však jedná o metody. Takto dekorovaná metoda je vždy asynchronní, tudíž v případě návratové hodnoty nevrací hodnotu samotnou, nýbrž objekt *Promise*. Pro získání samotné hodnoty z tohoto objektu slouží volání pomocí mechanismu *await*. Výraz await je použitelný pouze v asynchronních funkcích, z čehož vyplývá, že i komponentní metody s dekorátorem @Method lze volat jen asynchronně.

Za účelem zasílání zpráv ve formě událostí slouží dekorátor **@Event**. Tento dekorátor umožňuje v rámci vzoru *publish–subscribe* komponentám odesílat zprávy odběratelům bez čekání na odpověď či návratovou hodnotu. Dekorátor lze aplikovat na komponentní atribut typu *EventEmitter* a kromě jiných nastavení definuje především název události, na níž pak ostatní metody komponent v aplikaci mohou naslouchat. Samotné odeslání zprávy proběhne zavoláním metody *emit* na dekorovaný atribut; do odesílané zprávy lze vkládat i objekty či data.

Přirozeným doplňkem dekorátoru @Event je dekorátor **@Listen**. Jak už název napovídá, metoda s tímto dekorátorem naslouchá na událost se specifikovaným názvem. Ve výchozím stavu takovéto metody očekávají události pouze v rámci svých přímých následnických komponent, což lze v nastavení dekorátoru změnit na jakýkoliv cíl v modelu DOM.

StencilJS poskytuje i metody pro definici chování v rámci životního cyklu komponenty. Metoda render zmíněná na začátku této podsekce je volána při každé změně atributů komponenty. Tuto metodu obalují volání metod **componentWillRender** a **componentDidRender** pro specifikaci chování před či po vykreslení komponenty. Kromě dalších metod v životním cyklu komponenty lze ještě zmínit metody **componentWillLoad** a **componentDidLoad**, jež mají podobný význam jak předchozí zmíněná dvojice metod, akorát se jejich volání uplatňuje pouze před či po prvním render volání komponenty.

4.1.2 Struktura aplikace

Zdrojový kód projektu aplikace obsahuje jak Stencil komponenty, tak i další TypeScriptové soubory poskytující další funkcionality či datové modely v rámci přehledného a snadnějšího vývoje aplikace. Následovat bude přehled zdrojové struktury této aplikace, přičemž z přehledu budou vynechány takové strukturní jednotky, jež nebyly přidány či upraveny v rámci této diplomové práce. Tento přehled je následující:

- **components** – Komponenty vytvořené v rámci StencilJS API.
  - **field-editors** – Komponenty tvořící jednotlivá editační pole ve formuláři pro úpravu aktivit.
    - **expression-field** – Textové pole uzpůsobené pro víceřádkový vstup.
    - **checkbox-connection-list-field** – Zaškrtávací seznam sloužící ke zvolení relevantních vstupů pravidla ze seznamu všech vstupů.
    - **text-field** – Jednořádkové textové pole.
    - **text-field-multi** – Seznam sloužící k přidávání nových výstupů pravidla pomocí textového pole či jejich odebírání.
  - **workflow-designer** – Komponenty zajišťující funkcionalitu samotného modelovacího nástroje.
    - **designer** – Nastavení knihovny jsPlumb a vysílání událostí v rámci úprav pipeline do dalších komponent v aplikaci.
    - **designer-host** – Kontejner pro komponentu designer a komponenty modálních oken aplikace; zajišťuje základní funkcionalitu úprav pipeline.
    - **designer-wrapper** – Obaluje komponentu designer-host za účelem jednoduššího vložení a nastavení webové komponenty modelovacího nástroje v HTML hostitelské aplikace.
    - **import-export** – Poskytuje konverzi namodelované pipeline do Snakefile, JSON a dalších formátů a načítání takto exportovaných souborů zpět do nástroje.
    - **export-button** – Komponenta tlačítka pro exportování.
    - **save-load** – Komponenty zajišťující ukládání a načítání pipelines ze serveru.
      - **pipeline-loader** – Načítá pipeline ve formátu JSON uloženou na serveru.
      - **pipeline-saver** – Obstarává ukládání pipeline ve formátech JSON a Snakefile na server.
- **drivers** – Třídy implementující metodu pro nastavení dat zobrazovaných editačních polí a metodu pro aktualizaci stavových dat aktivity z odeslaného formuláře v rámci globálního workflow objektu; ke každému formulářovému poli z field-editors existuje odpovídající driver.

- **models** – Datové modely používané napříč aplikací v podobě rozhraní či vlastních datových typů.
- **plugins** – Třídy implementující metody pro definici aktivit k použití uživatelem.
  - **pipeline-activities** – Obsahuje definice pro aktivity představující Snakemake pravidla, bloky vstupů, Python blok a collector pravidla.
- **services** – Třídy definující funkcionalitu v rámci specifických částí aplikace.
  - **activity-manager** – Zajišťuje vykreslování bloku aktivity a metody pro vyhledávání aktivit ve stavovém workflow objektu.
  - **default-activity-handler** – Poskytuje metody pro nastavení vykreslování bloku aktivity.
  - **graph-checker** – Implementace algoritmů pro zjištění acyklicity a souvislosti grafu workflow před serializací do Snakefile.
  - **snake-serializer** – Serializační třída umožňující konverzi namodelované pipeline do souboru formátu Snakefile.
  - **snake-service** – Obsahuje metody pro filtrování relevantních vstupů pravidel a konverzi vstupů a výstupů pravidel do řetězců.

4.1.3 Bloky aktivit

Do aplikace bylo nutné přidat bloky aktivit reprezentující Snakemake pravidla, vstupní data, Python kód a collector pravidla. Za tímto účelem slouží v aplikaci rozhraní **WorkflowPlugin** a objekt **pluginStore**. Zmíněné rozhraní definuje metody pro vytvoření aktivit v rámci společné kategorie. V případě aktivit pro tvorbu pipelines má tato kategorie název *PipelineActivities*.

Samotná tvorba aktivit probíhá prostřednictvím datového modelu **ActivityDefinition**. Tento model obsahuje mj. název typu aktivity, název zobrazovaný v uživatelském rozhraní, popis aktivity či pole *properties* pro určení vlastností (a také nastavení odpovídajících formulářových polí), jež dané aktivitě náleží. Globální objekt pluginStore pak umožňuje vytvořenou kategorii přidat do aplikace. Ukázka tvorby aktivity Snakemake pravidla vypadá následovně:

```
1 private rule = (): ActivityDefinition => ({
2     type: 'Rule',
3     displayName: 'Rule',
4     description: 'Rule definition including its name,
5                   inputs, outputs and run commands.',
6     ...
7     properties: [
```

```
8           ...
9           {
10              name: 'inputs',
11              type: 'checkbox-connection-list',
12              label: 'Input',
13              hint: 'Inputs based on the incoming connections.'
14          },
15          {
16              name: 'outputsState',
17              type: 'text-multi',
18              label: 'Output',
19              hint: 'Output definitions for the current rule.'
20          },
21          ...
22      ]
23  });
```

Listing 4.1: Kód aktivity Snakemake pravidla

V rámci realizace řešení jsem se při tvorbě aktivity Snakemake pravidla rozhodl implementovat možnosti definice názvu pravidla, vstupů, výstupů a příkazů run, shell a log. Tato podoba pravidla v modelovacím nástroji umožní uživateli modelovat pipelines v základním rozsahu a bude poskytovat dostatečné možnosti pro modelování celé řady možných řešení.

4.1.4 Editory formulářových polí

Za účelem vytvoření editační funkcionality jednotlivých bloků aktivit v podobě, v jaké je popsána v podsekci 3.3.2 o návrhu uživatelského rozhraní, bylo třeba implementovat další formulářová pole. Proto jsem do aplikace přidal komponenty **TextFieldMulti** a **CheckboxConnectionListField** a přidal podporu textového editoru **CodeMirror**.

4.1.4.1 Textový editor CodeMirror

Místo použití obyčejného víceřádkového pole v editačních formulářích aktivit jsem usoudil, že by pro uživatele bylo pohodlnější k tvorbě Python/Snakemake výrazů používat nástroj se schopnostmi textového editoru. Za tímto účelem byla použita JavaScriptová knihovna CodeMirror, jež v prostředí webového prohlížeče poskytuje základní funkcionalitu běžných textových editorů kódu jako zvýrazňování syntaxe kódu řady podporovaných programovacích jazyků, číslování řádků či nastavitelné klávesové zkratky.

Ačkoliv CodeMirror nepodporuje syntaxi nástroje Snakemake, lze pro zvýrazňování syntaxe alespoň použít dostupné podpory jazyka Python. Samotný editor je možné přidat do webové stránky (nebo v případě této práce do komponenty) několika různými způsoby, ovšem já zvolil způsob použití metody knihovny *fromTextArea()*. Tato metoda umožňuje editor přidat nahrazením existujícího *textarea* elementu, přičemž tento původní element je pouze schován a lze jej dále používat k získání jeho hodnoty či odeslání ve formuláři.

Editor CodeMirror pak v rámci použití v této aplikaci synchronizuje své změny pomocí metody *save()* na instanci editoru *codeMirror*. Editor je použit v komponentách TextFieldMulti a ExpressionField, v obou případech je nastaven na zobrazování čísel řádků, automatické odsazování na novém řádku a o půl vteřiny pozdějším načítání z toho důvodu, aby se předešlo načtení editoru ještě dříve, než se zobrazí modální okno s formulářem; nepoužití tohoto přístupu má tendenci editor zobrazovat chybně. Nastavení CodeMirror editoru v komponentě vypadá takto:

```
1 this.codeMirror = CodeMirror.fromTextArea(this.outputTextField, {
2     mode: 'python',
3     lineNumbers: true,
4     smartIndent: false,
5     indentWithTabs: true,
6     tabSize: 2
7 });
8 this.codeMirror.on('change', () => this.codeMirror.save());
9 let self = this;
10 setTimeout(function() { self.codeMirror.refresh() }, 500);
```

Listing 4.2: Nastavení textového editoru CodeMirror

4.1.4.2 Komponenta TextFieldMulti

Komponenta TextFieldMulti představuje formulářový prvek umožňující přidávat či odebírat výstupy pravidla. Přidání výstupu probíhá prostřednictvím textového editoru CodeMirror. Komponenta registruje tlačítko *Add output*, jež po kliknutí vyvolá metodu pro přidání zadaného textového vstupu do seznamu výstupů pravidla. Již přidané výstupy lze odebrat kliknutím na tlačítko s křížkem přítomné vedle každé položky seznamu přidaných vstupů.

Každá položka tohoto seznamu má vnitřní podobu dvojice číselného identifikátoru vstupu a samotné textové hodnoty vstupu. Identifikátor je přiřazen na základě vnitřního čítače aktivity, jenž se zvýší pokaždé, když je přidán nový vstup do seznamu. Tento identifikátor je přidáván z důvodu využití v rámci komponenty CheckboxConnectionListField popsaného v následující podsekci.

V původní aplikaci vkládání HTML elementů formulářových komponent do modelu DOM odpovídajícími drivery probíhá skrze interpolovaný řetězec elementu. Kvůli tomuto důvodu a stejně

tak i kvůli aktualizaci dat aktivity po uložení formuláře prostřednictvím čtení formulářových dat (podporujících pouze primitivní datové typy jako číslo, boolean či řetězec) bylo nutné reprezentovat seznam výstupů v serializované podobě. Tento serializovaný řetězec je pak uložen v hodnotě skrytého formulářového pole.

Při přidávání či odebírání výstupu ze seznamu je tedy nutné nejprve deserializovat pole se seznamem, upravit jej dle požadované operace a následnou serializovanou hodnotu upraveného seznamu uložit jako novou hodnotu skrytého pole. Stencil pomocí JSX umožňuje provázání atributů komponenty (angl. *data binding*) s interpolovanými hodnotami těchto atributů v HTML elementech komponenty za použití @Prop dekorátoru.

Takto dekorovaný je i atribut *value* obsahující serializovaný řetězec seznamu výstupů pravidla. Úprava seznamu výstupů pak spočívá v nastavení atributu řetězce seznamu na novou hodnotu. Přidání či odebrání výstupu změnou tohoto atributu v rámci životního cyklu komponenty (popsaného v podsekci 3.2.6) vyvolá její nové vykreslení. Vzhledem k tomu, že vykreslování již přidaných vstupů se taktéž odráží od hodnoty zmíněného atributu, je tímto mechanismem zajištěno, že po změně v seznamu výstupů pravidla se HTML model seznamu automaticky aktualizuje.

4.1.4.3 Komponenta CheckboxConnectionListField

K poskytnutí možnosti vybírat relevantní vstupní data ze všech vstupních bloků nebo výstupů pravidel, jejichž hrany vedou do daného pravidla, vznikla komponenta CheckboxConnectionListField. Každý takový relevantní vstup je vykreslen jako zaškrtávací pole, jehož změna stavu vyvolá metodu pro přidání či odebrání vstupu ze seznamu zaškrtnutých vstupů.

Před samotným vykreslením komponenty je v rámci odpovídajícího driveru nutné nejprve vyhledat hrany vedoucí do daného bloku pravidla a následně prostřednictvím objektu *SnakeService* vyfiltrovat stávající seznam zvolených vstupů tak, aby v něm existovaly pouze relevantní vstupy. Pro daný vstup tak musí platit, že existuje ve výstupech daného zdrojového bloku aktivity a že ze zdrojového bloku vede do cílového bloku hrana. Tato skutečnost je zajištěna ukládáním zaškrtnutého pole vstupu jakožto dvojice identifikátoru zdrojové aktivity a identifikátoru vstupu v rámci seznamu vstupů zdrojové aktivity. Funkcionalita filtrování je implementována takto:

```
1 filterCheckedInputs(checkedInputs: ConnectionInput[],
2                     relevantConnections: Connection[], workflow: Workflow) {
3     return checkedInputs.filter(x => {
4         let connection = relevantConnections.find(y => y.sourceActivityId
5                                                     == x.activityId);
6         if (!connection) return false;
7         let sourceActivity = ActivityManager.findActivityById(
8             connection.sourceActivityId, workflow);
9         if (sourceActivity.type == "Rule") {
```

```
10              let outputsState: OutputActivityState = sourceActivity
11                  .state["outputsState"];
12              return outputsState.outputs.find(y => y.id == x.outputId);
13          }
14          return true;
15      });
16  }
```

Listing 4.3: Filtrování relevantních zaškrtnutých vstupů pravidla

Následné vykreslování vstupů a jejich úprava se podobně jako u komponenty TextFieldMulti odvíjí od hodnoty serializovaného seznamu relevantních aktivit a také serializovaného seznamu již zaškrtnutých vstupů. Před každým ze zobrazených výrazů vstupů je zobrazen i štítek obsahující název zdrojového pravidla či bloku vstupních dat. Štítky jsou barevně rozlišeny – výrazy pravidel jsou obarveny modrou a výrazy bloků vstupních dat zelenou barvou.

Při úpravě seznamu zaškrtnutých vstupů se v případě vstupu vedoucího z bloku vstupních dat jedná pouze o jediný vstup, a nikoliv o seznam vstupů (jak tomu je u aktivity pravidla), tudíž vstup nepotřebuje mít přiřazen žádný identifikátor. Toto je při ukládání zaškrtnutých vstupů ošetřeno přiřazením výchozí nulové hodnoty k identifikátoru vstupu.

4.1.4.4 Další úpravy editorů

Zmíněný životní cyklus komponenty uváděl do původní implementace editačního formuláře aplikace Elsa Designer chybu: v případě vyplnění a uložení editačního formuláře aktivity a následného otevření editačního formuláře jiné, dosud needitované aktivity docházelo k chybnému zobrazení formuláře. Prázdná data totiž neměla žádná data k vykreslení, proto se prostřednictvím životního cyklu komponenty nespustil její nový render a byl zobrazen formulář z předešlé editované aktivity.

V rámci všech editorů formulářových polí použitých v mé realizaci modelovacího nástroje jsem tak musel přidat mechanismus, jenž donutí všechny komponenty editačního formuláře k renderu při každém zobrazení formuláře. Tento mechanismus má podobu náhodného čísla, jež je před každým vykreslením každého formulářového pole vloženo do komponenty pole odpovídajícím driverem. Další úpravy formulářových editorů spočívaly v drobných úpravách jejich stylů a přidání podpory textového editoru CodeMirror do pole **ExpressionField**, použitého pro zadávání výrazu v bloku vstupních dat a také výrazů run, shell a log příkazů pravidla.

4.1.5 Konverze modelu pipeline do Snakefile

Do stávajících možností exportu namodelovaných pipelines bylo nutné přidat ještě podporu pro konverzi pipeline do její Snakefile podoby. Taková konverze ovšem podléhá několika omezením, jež

vyplývají z použití pipelines v nástroji Snakemake a také z návrhu uživatelského rozhraní popsaného v předchozí kapitole.

První takové omezení spočívá ve skutečnosti, že aby mohl být vygenerován výsledný Snakefile z namodelované pipeline, je nutné, aby graf pipeline (ze své povahy orientovaný) neobsahoval cykly. Nástroj Snakemake podporuje pouze acyklické grafy výpočetních úloh a při detekování cyklických závislostí úloh skončí běh nástroje s chybou. Proto je i před konverzí modelu pipeline do formátu Snakefile nutné zkontrolovat acyklicitu grafu modelu.

Druhé omezení vyplývá ze situace, kdy se v grafu pipeline vyskytují dvě nebo více částí, jež jsou vzájemně odděleny a nevede mezi nimi žádná hrana. V takovém případě může být problematické rozeznat, zda některé z těchto grafových komponent lze v konverzi vynechat, a především pak není jasná sémantika datových vztahů mezi pravidly dané pipeline. Takovýto graf modelu pipeline vůči konverzi na Snakefile není považován za validní.

Pro konverzi je tedy nezbytné, aby byl graf modelu souvislý. V rámci orientovaných grafů lze zjišťovat dva typy souvislosti: slabou a silnou. Po zahrnutí prvního omezení do kontroly validity grafu modelu před konverzí je možné u grafu modelu pipeline zjišťovat pouze souvislost slabou, tedy souvislost grafu symetrizovaného.

4.1.5.1 Kontrola acyklicity grafu modelu

Zjišťování acyklicity grafu jsem implementoval jako algoritmus využívající metodu hledající topologické uspořádání grafu. K orientovanému grafu lze nalézt jeho topologické uspořádání, právě když se jedná o graf typu DAG. Pro účely konverze pipeline není nutné nalézt konkrétní uspořádání, proto lze algoritmus zjednodušit tak, aby pouze dokazoval či vyvracel existenci topologického uspořádání nad daným grafem modelu. Pokud nějaké takové uspořádání bude existovat, pak lze prohlásit, že graf je acyklický.

Algoritmus začíná inicializací pole, jež bude obsahovat takové uzly grafu, do nichž nevedou žádné hrany. Poté, dokud toto pole není prázdné, odebere se jeden uzel z pole. Všechny hrany vedoucí z toho uzlu jsou následně z grafu odebrány. Do pole jsou přidány takoví potomci daného uzlu, již nemají žádné vstupní hrany. Algoritmus poté pokračuje v následující iteraci cyklu dalším odebráním uzlu z pole. Pokud po dokončení poslední iterace v grafu pořád existuje hrana, graf obsahuje cyklus; v opačném případě je acyklický. Má implementace tohoto algoritmu vypadá takto:

```
1 let nodesNoInNeighbors: GraphNode[] = [];
2 this.nodeIds.forEach(x => {
3     let node = this.nodesMap[x];
4     if (node.inNeighbors.length === 0) nodesNoInNeighbors.push(node);
5 });
6 while (nodesNoInNeighbors.length > 0) {
7     let node = nodesNoInNeighbors.pop();
```

```
8     node.outNeighbors.map(x => {
9         let neighbor = this.nodesMap[x];
10        neighbor.inNeighbors = neighbor.inNeighbors.filter(y => y != node.id);
11        if (neighbor.inNeighbors.length === 0)
12            nodesNoInNeighbors.push(neighbor);
13    });
14    node.outNeighbors = [];
15
16 let someEdgeExists = false;
17 this.nodeIds.map(x => {
18    const node = this.nodesMap[x];
19    if (node.outNeighbors.length > 0) someEdgeExists = true;
20 });
21 return !someEdgeExists;
22 }
```

Listing 4.4: Implementace kontroly acyklicity grafu

4.1.5.2 Kontrola souvislosti grafu modelu

Pro zjištění souvislosti grafu modelu byl použit algoritmus založený na procházení grafu do hloubky (*depth-first search* – DFS). Podstata algoritmu spočívá v procházení symetrické verze původního orientovaného grafu z libovolného počátečního uzlu. Pokud v této verzi grafu existuje alespoň jeden uzel, jehož nelze dosáhnout cestou z počátečního uzlu, není tato verze grafu souvislá a původní orientovaný graf není slabě souvislý. Opak souvislost grafu dokazuje.

Algoritmus nejdříve zvolí uzel, v němž průběh DFS bude začínat. Poté se nad tímto uzlem spustí rekurzivní DFS, jež při svém průchodu postupně označí jako navštívené všechny uzly dosažitelné z počátečního uzlu (včetně tohoto samotného uzlu). Nakonec se provede kontrola, při níž se zjistí, zda v grafu ještě existuje nějaký nenavštívený uzel, a pokud ano, pak původní graf není slabě souvislý; jinak platí opak.

4.1.5.3 Serializace do Snakefile

Po kontrole validity grafu namodelované pipeline z hlediska acyklicity a souvislosti grafu přichází na řadu samotná konverze modelu pipeline do její Snakefile podoby. Ta začíná aktualizací aktivit pravidel stejným způsobem jako u komponenty CheckboxConnectionListField (popsané v podsekci 4.1.4.3). Pro každé pravidlo či collector pravidlo se tedy vyfiltrují pouze aktuální vstupy pravidla.

Následuje přidání kódu z Python bloku do výstupního Snakefile řetězce, pokud takový blok v modelu konvertované pipeline existuje. Poté se pod stejnou podmínkou přidá i collector pravidlo

v podobě svého názvu a seznamu vstupů – k přidání dojde jen tehdy, pokud toto pravidlo definuje název a alespoň jeden vstup. V závěru přijde na řadu zbytek pravidel, kdy každé pravidlo je přidáno pouze v případě, že kromě názvu obsahuje alespoň jeden vstup a výstup. Při konverzi je nutné, aby serializační metoda dodržovala korektní odsazení textu v daných úrovních.

4.1.6 Vizuální notifikace

Do modelovacího nástroje bylo vhodné přidat funkcionalitu, jež by intuitivním způsobem zajišťovala zobrazování upozornění uživatele na různé chyby a také notifikací ohledně úspěšných operací. Za tímto účelem jsem v projektu použil knihovnu **Toastr**. Je to JavaScriptová knihovna umožňující zobrazovat tzv. *toast* notifikace, tedy stručné notifikace krátkého trvání. Toastr podporuje notifikace čtyř druhů: informativní notifikaci, varování, chybovou hlášku a notifikaci o úspěchu.

V aplikaci modelovacího nástroje jsou tyto notifikace použity v rámci operace konverze pipeline do formátu Snakefile a operací načítání a ukládání pipelines na server. Knihovna poskytuje možnosti pro nastavení pozice a pořadí notifikací, základních animací objevování či mizení notifikací, zobrazení pásu postupu časovače zavření notifikací či registrace naslouchače na kliknutí na notifikaci. Pro všechny notifikace jsem tedy nastavil desetivteřinový interval zobrazení notifikací v pravém dolním rohu, doprovázený pásem postupu časovače.

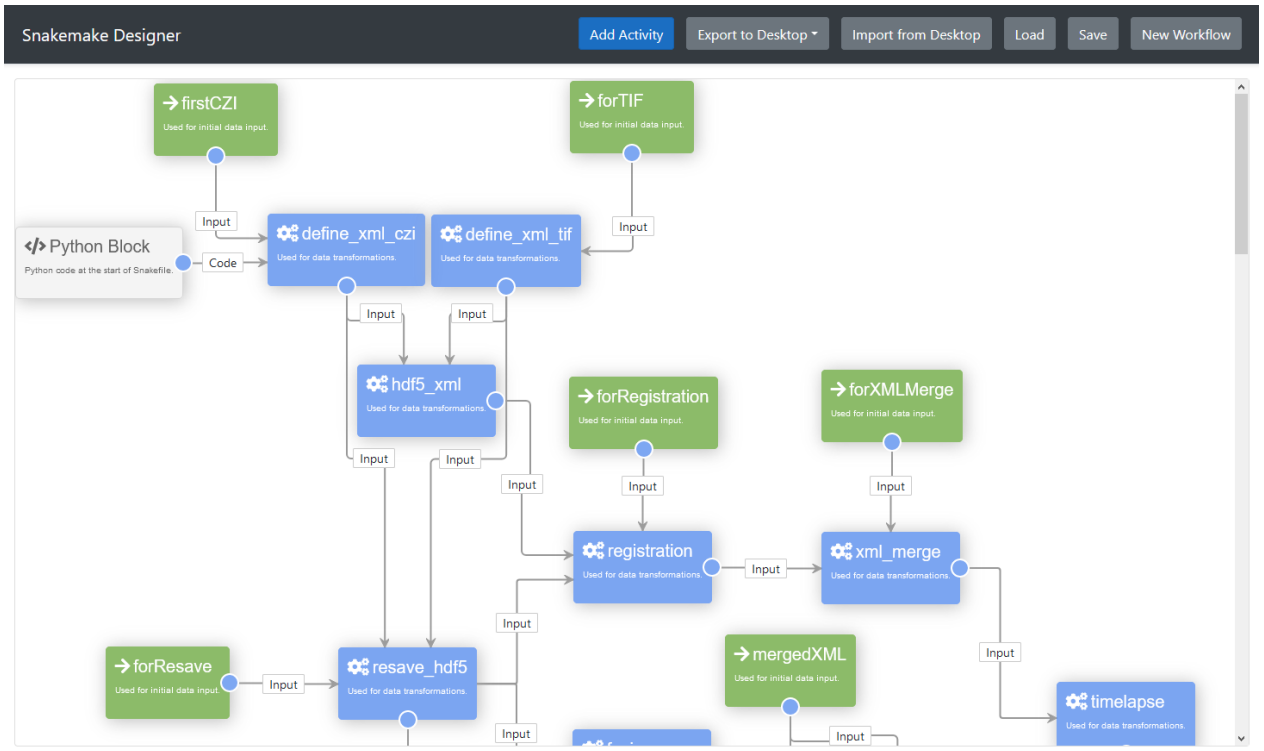
4.2 Výsledné uživatelské rozhraní

V této podsekci se zaměřuji na popis podoby výsledného uživatelského rozhraní a ovládání modelovacího nástroje. Výsledné rozhraní odráží jednak implementaci popsanou v předchozí sekci a jednak další implementační detaily, jež jsem do předchozí sekce ve prospěch stručnosti nezahrnul z důvodu vyhnutí se příliš drobnému detailnímu popisu. Tyto detaily tedy budou součástí této sekce z hlediska toho, jak se tyto detaily projevují ve výsledném uživatelském rozhraní.

4.2.1 Návrhové plátno nástroje

Po spuštění modelovacího nástroje v rámci webové stránky v prohlížeči se uživateli zobrazí návrhové plátno s ovládací nabídkou umístěnou v pravém horním rohu (viz obr. 4.1). Zpočátku se načte výchozí stav nástroje, kdy jsou na plátno vloženy dvě prázdné aktivity typu Snakemake pravidla. Uživatel může vytvářet vazby mezi aktivitami pomocí tažením modrého orbu zdrojové aktivity levým tlačítkem myši na blok cílové aktivity. Na vytvořené vazbě se zobrazí štítek, jenž popisuje význam vazby. Všechny vazby v nástroji jsou typu *Input* s výjimkou vazby vedoucí z bloku Python kódu – v takovém případě je typ vazby *Code*.

Vytvořenou vazbu lze zrušit kliknutím na štítek vazby. Uživatel může vést i více vazeb z jedné aktivity opětovným tažením modrého orbu dané aktivity. Nástroj nepodporuje tvorbu vazeb vedoucích z nějaké aktivity do té samé aktivity. Také nelze vytvořit vazbu vedoucí z nějaké aktivity **A** do



Obrázek 4.1: Výsledná podoba návrhového módu modelovacího nástroje

aktivity **B**, pokud již existuje vazba opačná, tedy vazba z **B** do **A**. Výše popsaná možnost odstraňování vazeb i omezení cyklických vazeb vyžadovaly úpravu nastavení knihovny jsPlumb v komponentě *designer*.

Hlavní nabídka na horní liště poskytuje tlačítka pro provedení základních operací nad modelem pipeline. Je zde možné vyvolat nabídku pro přidání nové aktivity do návrhového plátna, importovat či exportovat pipelines v rámci lokálního úložiště, ukládat či načítat pipelines skrze komunikaci se serverem nebo vyvolat tvorbu nové pipeline resetováním stavu nástroje na prázdné plátno.

Kromě hlavní nabídky je k dispozici i kontextové menu, jež je možné otevřít pravým kliknutím do plochy návrhového plátna. Při kliknutí do prázdného prostoru plátna lze přidat novou aktivitu. Pravým kliknutím na některou z aktivit na plátně může uživatel volit mezi operacemi úpravy aktivity a jejím odstranění. Editační mód aktivity lze také otevřít dvojitým kliknutím na danou aktivitu.

Bloky aktivit jsou na plátně barevně odlišeny podle svých typů: pravidla jsou modrá, bloky vstupních dat zelené, collector pravidlo je červené a blok s Python kódem šedý. S výjimkou Pythonovského bloku je u všech aktivit možnost definovat název (jenž je v případě pravidel a collector pravidla i povinný), který se pak zobrazí v bloku aktivity v návrhovém plátně místo nápisu značícího obecný typ aktivity. Vedle barvy, nápisu a krátkého popisu jsou bloky aktivit odlišeny i ilustrativními ikonami.

Obrázek 4.2: Výsledný editační formulář aktivity Snakemake pravidla – část 1.

4.2.2 Úprava aktivit

Přidání aktivity přes hlavní nabídku nebo kontextové menu zapříčiní zobrazení modálního okna s nabídkou dostupných aktivit. Po kliknutí na tlačítko *Select* u daného typu aktivity se vloží nová aktivita do návrhové plochy. Následně lze aktivitu upravovat otevřením modálního okna s editačním formulářem aktivity prostřednictvím dvojitého kliknutí či kontextového menu.

Editační formulář v jednotlivých polích zobrazí aktuální stav aktivity uložený ve stavovém objektu pipeline. Formulář po úpravách lze uložit tlačítkem *Save*; při stisknutí tlačítka *Cancel* nebo při kliknutí mimo modální okno formuláře se modální okno zavře bez ukládání změn. Kromě definice názvu (u aktivit pravidla, collector pravidla či bloku vstupních dat) uživatel všechen textový vstup zaznamenává prostřednictvím textového editoru CodeMirror.

Ten je nastavený na zobrazování čísel řádků, zvýrazňování Python syntaxe kódu a odsazení textu tabulátorem o šířce dvou mezer (při zachování znaku tabulátoru v konvertovaném Snakefile). Vstupy Snakemake pravidla uživatel volí v zaškrtávacím seznamu popsaného v podsekci 4.1.4.3. Výstupy pravidla jsou po zadání vstupu do textového pole přidávány tlačítkem *Add output* a následně odebírány příslušným červeným tlačítkem s ikonou křížku vedle daného výstupu v seznamu již přidaných výstupů. Ukázku vyplněného editačního formuláře lze nalézt na obr. 4.2 a 4.3.

Run

```
part_string = produce_string(
  """{fiji-prefix} {sysconfcpus} {num_cores_output} \
  {fiji-app} {memory-prefix}{mem_output} \
  -Dimage_file_directory={jdir} \
  -Dfirst_xml_filename={output_xml} \
  -Dhdf5_xml_filename={output_hdf5_xml} \
  -Dresave_angle={resave_angle} \
  -Dresave_channel={resave_channel} \
  -Dresave_illumination={resave_illumination} \
  -Dresave_timepoint={resave_timepoint} \
  -Dsubsampling_factors={subsampling_output} \
  -Dhdf5_chunk_sizes={chunk_sizes_output} \
  -Dtimepoints_per_partition={timepoints_per_partition} \
  -Dsetups_per_partition={setups_per_partition} \
  -Drun_only_job_number=0 \
  -Dfusion_switch={fusion_switch} \
  -Dconvert_32bit={convert_32bit}\
  -- --no-splash {path_bsh}""",
  config["common"],
  config["Fiji_resources"],
  config["hdf5_output"],
  config["resave_hdf5"],
  jdir=JOBDIR,
  output_xml=config["common"]["hdf5_xml_filename"].strip('\"') + "_output_define",
  output_hdf5_xml=config["common"]["fusion_switch"].strip('\"') + "_" + config["common"]["hdf5_xml_filename"].strip('\"'),
  path_bsh=os.path.join(config["common"]["bsh_directory"],config["hdf5_output"]["bsh_file_hdf5"]))

part_string += " > {log} 2>&1 && touch {output}"
shell(part_string)
```

Run command definition for the current rule.

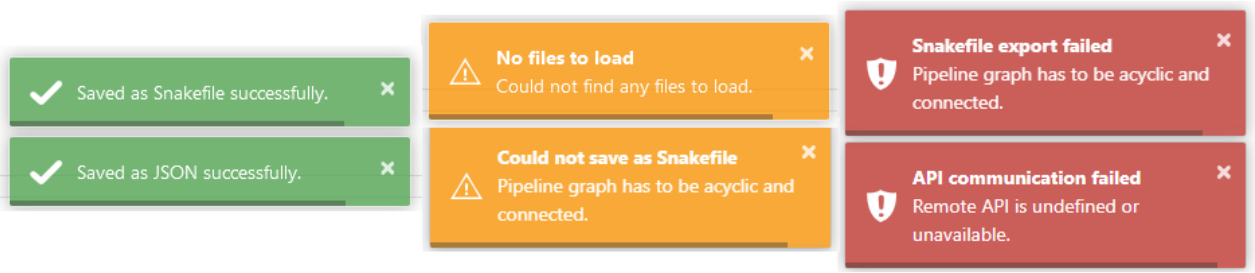
Shell

```
```

Shell command definition for the current rule.

Cancel Save

Obrázek 4.3: Výsledný editační formulář aktivity Snakemake pravidla – část 2.



Obrázek 4.4: Všechny toast notifikace použité v aplikaci modelovacího nástroje

4.2.3 Načítání a ukládání modelu pipeline

Export modelu pipeline lze provést využitím tlačítka v hlavní nabídce a zvolením požadovaného formátu, přičemž následně se otevře systémové okno umožňující uložení výsledného souboru na lokální úložiště. Stejně tak i při importu pipeline z lokálního úložiště stačí v systémové nabídce zvolit daný soubor. Nelze importovat pipeline ve formátu Snakefile, tudíž se musí jednat o jeden z ostatních editovatelných formátů.

Uložení pipeline na server lze docílit tlačítkem *Save* v hlavní nabídce. V modálním okně pak uživatel zadá název pipeline a potvrdí její uložení. Při načítání pipeline skrze tlačítko *Load* také dojde k vyvolání modálního okna, kde lze zvolit uloženou pipeline s požadovaným názvem a potvrdit načtení pipeline.

Konverze pipeline do formátu Snakefile a stejně tak i ukládání či načítání pipelines prostřednictvím komunikace se serverem zapříčiňuji zobrazování toast notifikací knihovny Toastr (zmiňované v podsekci 4.1.6), jež informují uživatele o výsledcích požadovaných operací. Zobrazení jednotlivých typů notifikací je rozděleno do těchto případů:

- **úspěch**:
  - export pipeline do formátu JSON a následné uložení na server proběhlo v pořádku,
  - export pipeline do formátu Snakefile a následné uložení na server proběhlo v pořádku,
- **varování**:
  - po zvolení možnosti načtení pipeline nejsou na serveru k dispozici žádné pipeline k načtení,
  - při ukládání pipeline na server nebylo možné model pipeline konvertovat do formátu Snakefile (graf modelu není acyklický a zároveň souvislý),
- **chyba**:
  - uživatel se pokusí o export nevalidního modelu pipeline do formátu Snakefile,
  - není možná komunikace s API serveru z důvodu technické závady (API nelze kontaktovat).

4.3 Integrace nástroje do systému WHL

Tato část kapitoly seznamuje čtenáře s realizací integrace implementovaného modelovacího nástroje Snakemake pipelines do systému WHL. Nejprve je zde popsán způsob, jakým byla webová komponenta nasazena v rámci systému WHL. Následovat pak budou popisy úprav jak na straně modelovacího nástroje, tak i na straně systému WHL za účelem zajištění komunikace mezi těmito dvěma aplikacemi.

4.3.1 Nasazení nástroje

Obecný postup nasazení webové komponenty vyvinuté pomocí nástroje StencilJS do webové aplikace mimo prostředí Node.js je následující:

1. sestavení webové komponenty do výsledných JavaScriptových souborů,
2. přidání a import externích závislostí webové komponenty do cílové webové aplikace,
3. přidání a import sestavených souborů do cílové webové aplikace,
4. vložení elementu webové komponenty do HTML stránky ve webové aplikaci.

Sestavení webové komponenty je zajištěno prostředím Node.js. Kompilace modelovacího nástroje má cílovou verzi jazyka nastavenu na ES2017, kód je tedy kompilován do modulů, jež se ve výsledné aplikaci na straně uživatele načítají po vzoru *lazy loadingu*. Sestavené soubory jsou pak vloženy do adresáře webových zdrojů systému WHL (adresář *wwwroot/js/snakemake-designer*).

Před importem modelovacího nástroje do HTML hlavičky webové aplikace je nutno nejprve přidat a importovat externí závislosti této webové komponenty. Adresář těchto závislostí má v systému WHL lokaci *wwwroot/js/lib*. Tyto externí závislosti tvoří Boostrap v4.3.1, jQuery v3.3.1, Toastr v2.1.4 a CodeMirror v5.59.4. Po importu stylů a skriptů těchto závislostí lze naimportovat samotný modelovací nástroj, přičemž stačí přidat hlavní skript *snakemake-designer.js* – ostatní skripty jsou importovány uvnitř tohoto souboru.

Následuje přidání elementu webové komponenty do HTML kódu webové stránky v systému. Vzhledem k tomu, že systém WHL je aplikací typu ASP.NET Core MVC, vytvořil jsem za tímto účelem nový *controller* a v něm novou akci *Designer* pro vykreslení stránky modelovacího nástroje. Bylo pak ještě nutné vytvořit stejnojmenný *view* obsahující samotné HTML včetně elementu modelovacího nástroje a jeho základního nastavení.

V původní aplikaci Elsa Designer vstupní element webové komponenty představovala komponenta *DesignerHost*. Tato komponenta ovšem nezahrnuje kompletní definici HTML struktury modelovacího nástroje – neobsahuje lištu s hlavní nabídkou ani kontejnery definující rozložení webové komponenty. Navíc tato komponenta poskytuje některé atributy nastavení, jež v rámci potřeb realizace této práce nejsou nutné.

Z těchto důvodů jsem v projektu aplikace modelovacího nástroje vytvořil ještě Stencil komponentu **DesignerWrapper**. Komponenta zjednodušuje přidání modelovacího nástroje do webové stránky, přičemž tvoří obálku komponenty DesignerHost. Přidává k ní také požadovanou strukturu kontejnerů (využívající frameworku Bootstrap) a horní lištu hlavní nabídky včetně provázání tlačítek nabídky s metodami komponenty DesignerHost. Poskytuje také jednodušší nastavení komponenty skrze atributy odpovídajícího HTML elementu. Přidání elementu webové komponenty a nastavení modelovacího nástroje ve své konečné podobě vypadá takto:

```
<wf-designer-wrapper
    canvas-height="300vh"
    designer-container-height="80vh"
    data-workflow='{
        "activities": [
          {
            "id": "rule123",
            "top": 100,
            "left": 100,
            "type": "Rule",
            "state": {}
          },
          {
            "id": "rule456",
            "top": 100,
            "left": 300,
            "type": "Rule",
            "state": {}
          }
        ],
        "connections": []
}' />
```

Listing 4.5: Výsledná podoba elementu obsahujícího modelovací nástroj

Nastavení *canvas-height* určuje celkovou výšku návrhového plátna, jež je nastavena na 300 % vnitřní výšky okna prohlížeče. Atribut *designer-container-height* představuje výšku elementu obsahujícího návrhové plátno; to se tedy na stránce roztáhne na výšku o této hodnotě a zbytek plátna lze v tomto elementu posouvat kolečkem myši. Posledním dostupným nastavením je *data-workflow*, jež definuje výchozí stav pipeline na návrhovém plátně. Jsou zde přidány dva bloky aktivit pravidel s určením polohy bloků, prázdným stavem aktivit a prázdným seznamem vazeb.

Pohled Designer v systému WHL tedy obsahuje takto nastavený element komponenty DesignerWrapper. Součástí dostupných nastavení této komponenty je i možnost definovat URL rozhraní API, skrze něhož komponenta komunikuje se serverem. Atribut specifikující toto nastavení je ale ve stránce Designer vynechán z toho důvodu, že kořenová adresa API na serveru je již z předešlého vývoje systému WHL uložena v JavaScriptové proměnné *baseURL*.

Tato stránka tedy obsahuje i skript, jenž URL rozhraní serveru komponentě nastaví po načtení stránky. Pohled využívá i objektu třídy **SignInManager** v rámci rozhraní *Identity*, jež zajišťuje, že uživatel má k dané stránce přístup pouze tehdy, pokud je přihlášen do systému WHL.

4.3.2 Komunikace se serverem ze strany nástroje

Za účelem ukládání a načítání pipelines pomocí úložiště na serveru jsem přidal komponenty **PipelineSaver** a **PipelineLoader**. Tyto komponenty jsou součástí komponenty DesignerHost, jež řídí zobrazování modálních oken těchto komponent a také případně nastavuje jejich vnitřní data.

4.3.2.1 Komponenta PipelineSaver

Tato komponenta obstarává ukládání pipeline exportované do formátů JSON a Snakefile na server. V HTML struktuře svého modálního okna definuje jednoduchý formulář se vstupním polem, do něhož uživatel zadává název své pipeline. Po uložení formuláře dojde k vyvolání samotné funkcionality zajišťující export pipeline a následnou komunikaci se serverem. Nejprve se skrze referenci na komponentu **ImportExport** vyvolá exportování pipeline do formátu JSON. Poté je vytvořen objekt formulářových dat obsahující exportovaný JSON a tento objekt je odeslán na API serveru metodou HTTP POST. Uživatel je následně toast notifikací informován o úspěchu či selhání této operace.

Za normálních podmínek tedy vždy dojde alespoň k úspěšnému uložení modelu pipeline ve formátu JSON. Pak je proveden pokus o konverzi pipeline do formátu Snakefile. Pokud konverze dopadla neúspěchem z důvodu nevalidního modelu pipeline, je o této skutečnosti uživatel notifikován. V opačném případě se vytvoří nová formulářová data se Snakefile výstupem konverze a data jsou stejně jako předešlým způsobem odeslána na server. Nakonec dojde k uzavření modálního okna komponenty.

4.3.2.2 Komponenta PipelineLoader

Načítání pipelines uložených na serveru ve formátu JSON obsluhuje komponenta PipelineLoader. Modální okno této komponenty obsahuje rozbalovací nabídku, v níž uživatel volí z pipelines již uložených na serveru. Před zobrazením okna komponenty je ovšem nutno komponentě tento seznam nastavit, což je zajištěno komponentou DesignerHost. Ta ve své metodě kontaktuje server GET požadavkem na API serveru a jako návratová data v odpovědi na tento požadavek očekává pole

názvů JSON souborů uložených na serveru. Tato data předá komponentě PipelineLoader a vyvolá zobrazení jejího modálního okna.

Poté, co uživatel vybere název požadovaného souboru ze seznamu a potvrdí načtení pipeline, dochází k odeslání GET požadavku na URI daného souboru specifikovaného svým názvem. V odpovědi na požadavek je očekáván JSON objekt požadované pipeline, jenž je opět prostřednictvím komponenty ImportExport importován do nástroje. O případném selhání komunikace nástroje s API serveru je uživatel také informován toast notifikací. V závěru operace je okno komponenty uzavřeno.

4.3.3 Serverové API pro komunikaci s nástrojem

HTTP požadavky přicházející ze strany modelovacího nástroje zpracovává API controller třídy **SnakemakeControllerAPI**. Tento controller je nastaven na zpracovávání požadavků přicházejících na URI */{adresa aplikace}/api/snakemake*. K obsluze příchozích požadavků controller poskytuje metody těchto typů (první část názvu každé metody značí typ obsluhovaného HTTP požadavku, druhá část v závorkách značí název konkrétního zdroje v API):

- `HttpGet("files")` – slouží k zjištění názvů všech souborů pipelines uložených v uživatelském adresáři.
- `HttpGet("json_files")` – vrací seznam názvů JSON souborů pipelines uložených v uživatelském adresáři.
- `HttpGet("files/{fileName}")` – poskytuje možnost získat konkrétní soubor z uživatelského adresáře.
- `HttpPost("files")` – ukládá soubor přibalený ve formulářových datech požadavku do uživatelského adresáře.

Každá z těchto metod před samotným zpracováním požadavku nejprve pomocí rozhraní **SignInManager** zjistí, zda je uživatel přihlášen. Pokud ano, pak je přihlašovací jméno uživatele spolu s daty požadavku předáno volání jedné z metod služby třídy **SnakemakeHandler**, jež zajišťuje vnitřní logiku zpracování jednotlivých typů požadavků. Služba v rámci každého zpracování požadavků zajišťuje existenci adresáře obsahujícího uživatelské úložné prostory (definovaného v konfiguraci aplikace) a pak také existenci jednotlivých uživatelských adresářů uvnitř tohoto adresáře, pojmenovaných stejným názvem jako uživatelská jména odpovídajících uživatelů.

Ke každé metodě z výčtu výše pak existuje metoda ve službě SnakemakeHandler, jež implementuje požadovanou operaci nad uživatelským úložištěm na serveru. Pro názornost popisu funkcionality API na serveru systému WHL přikládám implementaci obsluhy požadavku na konkrétní soubor API controllerem a následně i službou SnakemakeHandler:

```
1 // SnakemakeControllerAPI
2 [HttpGet("files/{fileName}")]
3 public IActionResult GetFile([FromRoute] string fileName) {
4     try {
5         if (_signInManager.IsSignedIn(User)) {
6             var file = _snakemakeHandler.GetFile(getUserName(), fileName);
7             return Content(file, "application/json");
8         }
9         throw new Exception("User not signed.");
10    } catch (Exception e) {
11        _logger.LogError(e.ToString());
12        return StatusCode(500);
13    }
14 }
15
16 // SnakemakeHandler
17 public string GetFile(string userName, string fileName) {
18    string userPath = createUserDir(userName);
19    string filePath = userPath + "/" + fileName;
20    if (File.Exists(filePath)) return File.ReadAllText(filePath);
21    else return null;
22 }
```

Listing 4.6: Kód obsluhy požadavku na soubor v rámci Snakemake API v systému WHL

4.4 Prototyp funkcionality spouštění pipelines

V této závěrečné části kapitoly o realizaci řešení se zaměřím na tvorbu prototypu funkcionality pro spouštění pipelines namodelovaných v realizovaném modelovacím nástroji v rámci systému WHL. Účelem tohoto prototypu je především demonstrovat průběh komunikace mezi systémem WHL a vytvořeným mock-up middlewarem (a potažmo i middlewarem HEAppE) a také poskytnout základ pro následné rozšíření a dokončení této funkcionality na straně backendu systému WHL.

4.4.1 Mock-up verze API middlewaru

Vytvořená mock-up verze API middlewaru slouží k nahrazení funkcionalit, které ještě nejsou implementovány či otestovány v systému WHL, případně na straně middlewaru HEAppE. Mock-up poskytuje předem připravené odpovědi na malou sadu HTTP požadavků, jejichž implementace je

nutná ze strany WHL backendu pro vytvoření základního prototypu funkcionality spouštění pipelines (a potažmo výpočetních úloh obecně) na clusteru skrze daný middleware. Podoba mock-up API rozhraní souhlasí se specifikací API middlewaru HEAppE; u všech zahrnutých funkcí zpracovávajících požadavky odpovídají namapovaná URI jednotlivým adresám v rámci specifikace a odpovědi ve formě serializovaných JSON objektů mají příslušné struktury.

Tuto zástupnou verzi API middlewaru jsem se rozhodl implementovat pomocí mikro-frameworku **Flask** [25], a to především z důvodu rychlé a jednoduché implementace. Tento webový framework poskytuje možnosti pro tvorbu serverových aplikací v jazyce Python a spolu s šablonovacím nástrojem **Jinja** umožňuje vytvářet webová řešení pokrývající backend i frontend aplikace. V rámci této diplomové práce jsem ovšem použil pouze základní serverové možnosti tohoto frameworku. Výsledná mock-up verze rozhraní pak poskytuje možnosti zpracování požadavků prostřednictvím těchto funkcí:

- `authenticate_user_password()` – ve specifikaci slouží k autentizaci uživatele před jakoukoliv další komunikací s middlewarem, v odpovědi vrací autentizační token.
- `create_job()` – zakládá novou úlohu dle vstupní specifikace úlohy.
- `submit_job()` – předá založenou úlohu ke spuštění na clusteru.
- `get_current_info_for_job()` – zjistí stav založené úlohy (zda již úloha byla spuštěná, dokončená nebo má případně jiný stav).
- `get_file_transfer_method()` – pro danou založenou úlohu poskytne informace nutné k nahrání či stažení vstupních či výstupních dat úlohy.

Zde je ještě nutno poznamenat, že popisy účelů těchto funkcí slouží k informování čtenáře o jejich eventuálním významu, v rámci mock-upu se však úkony těchto funkcí nevykonávají a funkce tak především představují vstupní body pro zpracování požadavků příchozích z WHL backendu a poskytnutí validních odpovědí. V případě zpracování požadavku pro zjištění stavu úlohy kromě pouhého vrácení odpovědi také funkce simuluje běh spuštěné úlohy tak, že na každý n-tý příchozí požadavek (kdy n je určeno globální konstantou) odpoví stavem úlohy "dokončená" a na ostatní požadavky stavem "spuštěná". Úloha tudíž není na straně mock-upu middlewaru ve skutečnosti spouštěná. Do kódu funkce pro zpracování požadavku na zjištění stavu úlohy lze nahlédnout zde:

```
1 @app.route("/JobManagement/GetCurrentInfoForJob", methods=['POST'])
2 def get_current_info_for_job():
3     global info_counter
4     info_counter += 1
5     state = 8
6     if info_counter >= info_count_max_before_finish:
```

```
7           state = 16
8           info_counter = 0
9       return {
10          "id": request_json["submittedJobInfoId"],
11          "name": "name",
12          "state": state
13      }
```

Listing 4.7: Zpracování požadavku pro zjištění stavu úlohy mock-up verzí API middlewaru

4.4.2 Serverové API pro komunikaci s WHL frontendem a mock-up rozhraním

Podobně jako v podsekci 4.3.3 o serverové API zajišťující komunikaci s modelovacím nástrojem bylo i v případě komunikace backendu s frontendem a mock-up verzí API middlewaru nutné poskytnout backendové komunikační rozhraní. Tuto komunikaci v rámci prototypu funkcionality spouštění pipelines zajišťuje API controller **SnakemakeJobControllerAPI**. Controller obsluhuje požadavky na URI */{adresa aplikace}/api/snakemake_job* a poskytuje možnosti zpracování požadavků těchto typů (první část názvu – HTTP požadavku, část v závorkách – název zdroje v API):

- `HttpGet("pipelines")` – vrací seznam pipelines exportovaných do Snakefile formátu, jež byly modelovacím nástrojem uloženy do uživatelského adresáře daného uživatele.
- `HttpGet("user_files")` – poskytuje seznam souborů nahraných uživatelem pomocí WHL funkcionality nahrávání dat.
- `HttpPost("create_submit")` – vytvoří specifikaci úlohy obsahující mj. parametry šablony úlohy, jež jsou tvořeny názvem spouštěné Snakemake pipeline a poté názvy vstupních souborů pipeline; specifikaci úlohy odešle na middleware, zajistí přenos vstupních dat na úložiště clusteru a spuštění úlohy. Frontendu vrátí odpověď obsahující informaci o stavu vytvořené úlohy.
- `HttpGet("update_job")` – skrze volání API middlewaru zjistí stav dané úlohy, výsledek vrací frontendu; pokud je úloha ve stavu "dokončená", zajistí i přenos výstupních dat úlohy z úložiště clusteru.
- `HttpGet("job_output")` – vrací seznam výstupních souborů dokončené úlohy.
- `HttpPost("download_output")` – vytvoří archiv obsahující zvolené výstupní soubory dané úlohy a vrací *filestream* pro stažení vytvořeného archivu.

Za účelem odzkoušení prototypu funkcionality spouštění pipelines nejsou na backendu implementovány metody zajišťující datový přenos mezi backendem a úložištěm na clusteru, místo toho

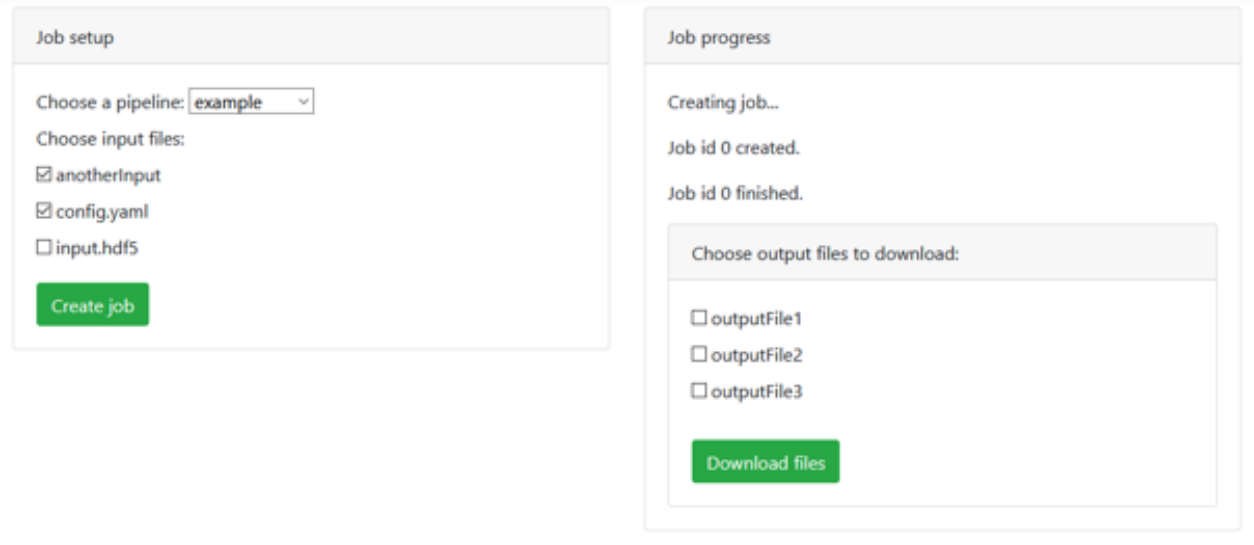
je v případě výstupních dat úlohy vytvářeno pár souborů k ověření funkčnosti tvorby výsledného archivu a jeho následného stažení uživatelem.

Controller pro komunikaci s middlewarem využívá metod třídy **RestAPIClient**, která zajišťuje serializaci obsahu požadavků do podoby řetězců JSON objektů, odesílání požadavků na danou adresu rozhraní middlewaru a následnou konverzi odpovědí zpět do objektů tříd jazyka C#. Každému sledu požadavků na middleware předchází požadavek controlleru na middleware k autentizaci za účelem získání řetězce s autentizačním tokenem (ačkoliv mock-up verze rozhraní samotnou autentizaci a vygenerování tokenu neimplementuje).

4.4.3 Výsledný prototyp funkcionality

Výsledná podoba realizovaného prototypu funkcionality spouštění pipelines prostupuje frontendovou i backendovou částí systému WHL v kombinaci s mock-up verzí rozhraní middlewaru. Prototyp stanovuje základní postup při spouštění pipelines uživatelem a využívá komunikace mezi backendem a mock-up verzí rozhraní. Jednotlivé kroky v rámci realizovaného prototypu funkcionality zachycuje následující scénář použití:

1. Přihlášený uživatel vstoupí na webovou stránku obsahující formulář pro nastavení úlohy ke spuštění.
2. Frontend webové stránky načte z backendu seznam uživatelem uložených pipelines a nahraných souborů.
3. Uživatel vybere ze seznamu uložených pipelines požadovanou pipeline ke spuštění.
4. Uživatel vybere ze zaškrtávacího seznamu nahraných souborů požadované vstupní soubory pipeline.
5. Uživatel potvrdí vytvoření a spuštění úlohy; konfigurace úlohy je odeslána na backend.
6. Backend z přijaté konfigurace vytvoří specifikaci úlohy, kterou předá middlewaru v požadavku na vytvoření úlohy.
7. Backend si od middlewaru vyžádá informace pro zahájení datového přenosu vstupních dat specifikovaných v konfiguraci úlohy.
8. Backend na middleware zašle požadavek na spuštění úlohy; frontendu je vrácena odpověď se stavem úlohy.
9. Frontend periodicky kontroluje stav úlohy skrze požadavky na backend; ten v rámci kontroly stavu úlohy odešle požadavek na rozhraní middlewaru a vrácenou odpověď předá zpět frontendu.



Obrázek 4.5: Výsledná podoba webové stránky pro prototyp funkcionality spouštění pipelines

10. Pokud je v rámci kontroly stavu úlohy na backendu zjištěn stav úlohy "dokončená", vyžádá si backend od middlewaru informace pro zahájení datového přenosu vstupních dat; v rámci prototypu nejsou následně data přenášena, nýbrž je na serverovém úložišti vytvořeno několik prázdných souborů představujících výstup pipeline.

11. Po zjištění stavu úlohy "dokončená" v rámci kontroly na frontendu načte frontend stránky z backendu seznam výstupních souborů.

12. Uživatel vybere ze zaškrtávacího seznamu výstupních souborů požadované soubory ke stažení; po potvrzení odeslání formuláře je seznam požadovaných souborů odeslán na backend.

13. Backend vytvoří archiv obsahující požadované výstupní soubory a frontendu vrátí odpověď obsahující filestream ke stažení archivu.

14. Frontend vyvolá uživatelskou interakci, v rámci níž může uživatel archiv uložit.

Frontendová část zajišťující uživatelskou interakci a komunikaci s backendem je tvořena webovou stránkou definující základní rozložení uživatelského rozhraní (viz obr. 4.5) a skripty v jazyce JavaScript. Ty obstarávají odesílání požadavků na backend a zpracovávání odpovědí včetně dynamické tvorby formuláře konfigurace pipeline (obsahující seznamy pipelines a vstupních souborů), zobrazování informací o stavu úlohy či tvorby formuláře pro zvolení výstupních dat ke stažení. Zde přikládám ukázku kódu pro vytvoření a spuštění úlohy ze strany frontendu:

```
1 var pipelineName = $('#snakemake-pipeline').val();
2 var inputFileNames = $('#input-files-checkboxes...').map(function () {
3     return $(this).val();
```

```
4 }).get();
5 $('#snakemake-job-progress').append('<p>Creating job...</p>');
6 $.ajax({
7     type: 'POST',
8     url: global_baseUrl + 'api/snakemake_job/create_submit',
9     data: JSON.stringify({ pipelineName, inputFileNames }),
10    contentType: 'application/json',
11 }).done(function (data) {
12    if (data.state == 0) {
13        setTimeout(snakemakeUpdateJob, 2000);
14        $('#snakemake-job-progress').append('<p>Job id ' + data.id +
15                                            ' created.</p>');
16    }
17 });
```

Listing 4.8: Skript pro vytvoření a spuštění úlohy z webového rozhraní

Kapitola 5

Testování

Kromě samotného návrhu a realizace řešení bylo potřeba v rámci této diplomové práce i výsledné řešení otestovat. Právě tímto aspektem se zabývá tato kapitola. Bude zde nejprve popsán způsob, jakým jsem k testování řešení přistoupil. Poté budou představeny jednotlivé postupy a technologie, jakými se testování řídilo. Součástí kapitoly budou i následné testovací scénáře s přehledy jednotlivých testů, jejich vstupních dat a výsledků testování.

5.1 Přístup k testování

Před samotným návrhem testovacích procesů je nutné zjistit, jaké testovatelné celky ve výsledném řešení vznikly a jakými způsoby lze tyto celky testovat. V rámci řešení vznikly tři samostatné celky. Prvním z nich je JavaScriptová webová komponenta představující samotný návrhový nástroj. Druhým celkem je pak rozšíření systému WHL poskytující rozhraní, se kterým může webová komponenta komunikovat za účelem načítání či ukládání pipelines vytvořených v modelovacím nástroji, a také rozšíření systému WHL o prototyp funkcionality spouštění pipelines. Poslední celek pak tvoří mock-up verze rozhraní middlewaru pro doplnění tohoto prototypu.

Tyto celky lze samostatně či společně testovat různými typy testů. Vzhledem k tomu, že se webová komponenta modelovacího nástroje i systém WHL skládají z dílčích komponent, naskýtá se možnost testovat komponenty těchto celků v rámci jejich samostatné funkčnosti pomocí *unit* testů. Dalším způsobem testování by pak mj. mohlo být např. integrační testování, jež by ověřovalo korektní komunikaci mezi dvěma zmiňovanými realizovanými částmi řešení. Také se nabízí možnost automatizovaného testování uživatelského rozhraní strojovým ovládáním webového prohlížeče.

Z hlediska unit testování webová komponenta modelovacího nástroje obsahuje Stencil komponenty, jež kromě samotného vykreslování obsahu poskytují také specifické funkcionality zajišťující dynamickou interakci s uživatelem. Vedle těchto komponent jsou součástí modelovacího nástroje i služby jako SnakeSerializer nebo GraphChecker sloužící ke konverzi namodelované pipeline do formátu Snakefile či ke kontrole podmínek, potřebných k uskutečnění této konverze.

Integrační testování komunikace mezi webovou komponentou a systémem WHL by znamenalo ověřování správnosti odpovědí API systému WHL na konkrétní typy požadavků přicházejících ze strany webové komponenty. Vzhledem k tomu, že komunikace mezi těmito dvěma částmi není moc rozsáhlá, je možné tento typ testování nahradit dostatečným unit testováním serverového API rozhraní systému WHL. Integrační testování komunikace by tak nemělo žádný zásadnější přínos. Podobně i u komunikace mezi WHL backendem a mock-up verzí middlewaru jsem se rozhodl otestovat příslušnou API na straně backendu systému WHL pomocí unit testů.

Automatizované testování uživatelského rozhraní by mohlo probíhat pomocí skriptů řídících chování webového prohlížeče. Tyto skripty by simulovaly uživatelský vstup a ověřovaly by tak správné reakce aplikace na různé úkony uživatele. Tyto úkony ovšem v rámci webové aplikace modelovacího nástroje nejsou příliš různorodé, proto by bylo vhodnější kontrolovat změny vnitřních stavů daných komponent v reakci na úkony simulované uvnitř unit testů.

Z těchto důvodů jsem se rozhodl plně se zaměřit na unit testování nejzásadnějších komponent jak na straně aplikace modelovacího nástroje, tak i na straně systému WHL. Celek tvořený mock-up verzí rozhraní middlewaru jsem do testování nezahrnul, jelikož tato mock-up verze vznikla pouze za účelem ověření prototypu funkcionality spouštění pipelines na straně WHL backendu. Kromě toho také mock-up rozhraní nemá příliš rozsáhlé možnosti funkcionality.

5.2 Testování modelovacího nástroje

Za účelem efektivního a automatizovaného unit testování je vhodné využít testovacího nástroje či frameworku, jež usnadňuje tvorbu a spouštění testů a kontrolu jejich výsledků. Nástroj StencilJS k zajištění testovacích schopností používá framework Jest, určený k testování JavaScriptových projektů [26]. Tento framework umožňuje tvorbu a spouštění testů včetně možnosti tvorby mock objektů, různorodých způsobů kontroly očekávaných výsledků či tvorby reportů pokrytí kódu po skončení běhu testů.

StencilJS pak poskytuje API, pomocí něhož lze Stencil komponenty testovat skrze Jest ve dvou různých režimech: *spec* testování a *end-to-end* (zkráceně E2E) testování. Spec testy jsou spouštěny v prostředí Node.js a představují analogii k unit testům, poskytují tedy možnosti pro izolované testování Stencil komponent i obecných JavaScriptových třídních objektů a funkcí. Specificky pro testování Stencil komponent pak lze v rámci spec testů využít i omezeného simulovaného webového prostředí, pomocí něhož je možné ověřovat, zda se komponenty a jejich HTML vykreslují správně.

Testování E2E poskytuje možnosti testovat Stencil komponenty a interakce mezi vícero komponentami v kontrolovaném prostředí prohlížeče Chrome spouštěného v *headless* režimu (tzn. v režimu bez GUI). V tomto režimu je pak možné mj. manipulovat s vykresleným DOM modelem, zadávat textový vstup do formulářových polí či přecházet na různá URL v rámci testované StencilJS aplikace. Tento režim má ovšem zásadní nevýhodu v tom, že nelze využít možnosti tvorby mock objektů. Je

tak problematické plnění externích závislostí aplikace, jež nejsou součástí kompilovaného výstupu. Proto jsem se k testování modelovacího nástroje rozhodl použít pouze spec testů.

5.2.1 Tvorba testů

Každý test vytvořený prostřednictvím frameworku Jest je definován voláním metody s názvem **test** (či s aliasem **it**). Do argumentu této metody lze vložit popis, pod jakým bude reportován výsledek testu po jeho ukončení. Kromě této základní podoby testu lze využít i tvorby parametrizovaných testů, kdy na jeden testový postup je možné aplikovat více n-tic vstupních dat. Toho lze docílit voláním rozšiřující metody **it.each**, jejíž první argument definuje jednotlivé vstupní sady dat. Ve specifikaci popisu testových instancí pak lze použít formátovací značky, jež pro každou konkrétní instanci do jejího popisu doplní až n hodnot z odpovídající vstupní n-tice.

Samotný vnitřní kód testu pak obvykle obsahuje další přípravu vstupních dat a stavu testované komponenty, provádění operací komponenty potřebných k otestování a následné kontroly výstupních hodnot a konečného stavu komponenty. V případě spec testování Stencil komponent je součástí přípravy dat i vytvoření **SpecPage** instance, jež je schopna ve zjednodušené podobě simulovat vykreslování komponent ve webovém prostředí. Tento objekt pak obsahuje výsledné HTML komponenty spolu s objektovou instancí komponenty. Prostřednictvím SpecPage tak lze volat veřejné metody testované komponenty a přistupovat k jejím atributům. Pro názornost zde uvádím ukázku kódu jednoho z vytvořených testů:

```
1  describe('testing non-addible values', () => {
2      it.each([
3          ['empty', '', 0],
4          ['null', null, 0],
5          ['undefined', undefined, 0]
6      ])('should not add %s output', async (inputVarDesc: string, inputVar: any,
7                                            expected: number) => {
8          const value = JsonParserUtils.stringifySingleQuoteSafe(
9              { idCounter: 0, outputs: [] });
10         const page = await newSpecPage({
11             components: [TextFieldMulti],
12             html: `<wf-text-field-multi value='${value}'></wf-text-field-multi>`
13         });
14
15         page.rootInstance.outputTextField = { value: inputVar };
16         await page.waitForChanges();
17
18         page.rootInstance.addOutput(new MouseEvent('click'));
```

```
19          const parsed = JSON.parse(page.rootInstance.value);
20          expect(parsed.outputs).toHaveLength(expected);
21      });
22  });
```

Listing 5.1: Ukázka testu vytvořeného ve frameworku Jest

Vedle samotných testů je možné také definovat i metody, jež se váží na životní cyklus testů a testových sad. Těmito metodami jsou samopopisné metody **beforeAll**, **beforeEach**, **AfterAll** a **AfterEach** a jejich parametrem je vždy funkce, jež má být v dané fázi testu zavolána. Tato funkce pak může obstarávat inicializaci proměnných, konfiguraci testového prostředí či obnovení původního stavu objektů.

Framework Jest obsahuje i API pro tvorbu mock objektů, tedy objektů simulujících chování vzorových objektů s možností úprav tohoto chování pro účely testování. Metodám mock objektů v rámci tohoto frameworku tak lze nastavovat vlastní implementace nebo definovat konkrétní návratové hodnoty či posloupnosti návratových hodnot. Součástí vnitřní definice testu pak může být jak tvorba takovýchto objektů, tak i následná kontrola, zda byla konkrétní metoda mock objektu zavolána a případně kolikrát či s jakými argumenty.

5.2.2 Testovací scénáře

Při testování webové komponenty modelovacího nástroje jsem se zaměřil na testování těch nejzásadnějších částí nástroje, jejichž výpočetní logika by mohla skýtat největší množství chyb. Za tímto účelem vznikly čtyři testovací scénáře, jež definují sady testovacích případů pro formulářové funkcionality přidávání a odebírání vstupů a výstupů Snakemake pravidel, kontroly souvislosti a acyklicity grafu pipeline a konverzi modelu pipeline do Snakefile formátu.

Součástí každého testovacího scénáře jsou testovací případy ověřující funkčnost komponent za běžných i výjimečných podmínek. Jednotlivé scénáře jsou popsány v následujícím výčtu. Každý scénář obsahuje souhrnný popis dané testované funkcionality a pak i popisy samotných testovacích případů scénáře. Konkrétní informace o vstupních datech případů, jejich očekávaných výstupech a výsledcích jsou zaznamenány v příslušných tabulkách.

Je nutno poznamenat, že jeden testovací případ zde může být spouštěn s různými variacemi vstupních dat a počátečních stavů komponenty. V takových situacích jsou pak tato testová spuštění v tabulkách testovacích scénářů zahrnuta pod dané testovací případy. Tabulka pro testovací scénář I je pro názornost zahrnuta v samotném textu pod výčtem scénářů níže, ostatní tabulky scénářů pak lze nalézt v příloze A. Jedná se tedy o tyto scénáře:

- **Testovací scénář I:**
  - **testovaná funkcionalita**: přidávání či odebírání výstupů Snakemake pravidel skrze formulářový prvek – komponenta **TextFieldMulti**

  - **tabulka testovacích případů**: tab. 5.1
  - **testovací případy**:
    - **Případ 1** – kontrola správného vykreslení komponenty za podmínky korektního nastavení parametru stavového objektu výstupů
    - **Případ 2** – kontrola správného vykreslení komponenty ve výjimečných situacích (chybné nastavení parametru)
    - **Případ 3** – ověření funkcionality přidávání výstupů po dosazení nevalidních hodnot
    - **Případ 4** – ověření funkcionality přidávání výstupů po dosazení validních hodnot
    - **Případ 5** – ověření funkcionality odebírání výstupů po dosazení nevalidních hodnot
    - **Případ 6** – ověření funkcionality odebírání výstupů po dosazení validních hodnot

- **Testovací scénář II:**
  - **testovaná funkcionalita**: přidávání či odebírání vstupů Snakemake pravidel skrze formulářový prvek – komponenta **CheckboxConnectionListField**
  - **tabulky testovacích případů**: tab. A.1 a A.2 (viz příloha A)
  - **testovací případy**:
    - **Případ 7** – kontrola správného vykreslení komponenty za podmínky korektního nastavení parametrů stavového objektu vstupů a objektu aktivit s vazbami na konkrétní pravidlo
    - **Případ 8** – kontrola správného vykreslení komponenty ve výjimečných případech (chybné nastavení parametrů)
    - **Případ 9** – ověření funkcionality přidávání vstupů po dosazení nevalidních hodnot
    - **Případ 10** – ověření funkcionality přidávání vstupů po dosazení validních hodnot
    - **Případ 11** – ověření funkcionality odebírání vstupů po dosazení nevalidních hodnot
    - **Případ 12** – ověření funkcionality odebírání vstupů po dosazení validních hodnot

- **Testovací scénář III:**
  - **testovaná funkcionalita**: ověřování validity grafu namodelované pipeline z hlediska acyklicity a souvislosti grafu – služba **GraphChecker**
  - **tabulka testovacích případů**: tab. A.3 (viz příloha A)
  - **testovací případy**:
    - **Případ 13** – ověření funkcionality kontroly acyklicity grafu ve výjimečných případech (dosazení chybné hodnoty)
    - **Případ 14** – ověření funkcionality kontroly acyklicity grafu v případech, že je graf acyklický

        - **Případ 15** – ověření funkcionality kontroly acyklicity grafu v případech, že graf obsahuje cyklus
        - **Případ 16** – ověření funkcionality kontroly souvislosti grafu ve výjimečných případech (dosazení chybné hodnoty)
        - **Případ 17** – ověření funkcionality kontroly souvislosti grafu v případech, že graf je souvislý
        - **Případ 18** – ověření funkcionality kontroly souvislosti grafu v případech, že graf není souvislý

- **Testovací scénář IV:**
    - **testovaná funkcionalita**: konverze namodelované pipeline do formátu Snakefile – služba **SnakeSerializer**
    - **tabulky testovacích případů**: tab. A.4 a A.5 (viz příloha A)
    - **testovací případy**:
        - **Případ 19** – ověření funkcionality konverze modelu ve výjimečných případech (dosazení chybné hodnoty)
        - **Případ 20** – ověření funkcionality konverze modelu při dosazení validní hodnoty, jež je ekvivalentní s prázdným výsledným řetězcem
        - **Případ 21** – ověření funkcionality konverze modelu při dosazení validní hodnoty, jež je ekvivalentní s neprázdným výsledným řetězcem

5.2.3 Vyhodnocení testování

Testovací scénáře *I–IV* měly v rámci konkrétních spuštění testů jednotlivých případů úspěšnost 72,5 %. Všechny neúspěchy byly zapříčiněny neošetřením krajních vstupních hodnot, což vyústilo buďto v selhání a ukončení běhu testu ještě před kontrolou konečných výsledků, nebo v nedefinované chování, jež se projevilo až ve výsledných asercích.

Jedinou výjimku tvoří testovací případ č. 21, kterým byl odhalen nedostatek funkcionality konverze pipeline do formátu Snakefile. V případě pokusu o serializaci víceřádkového výrazu výstupu pravidla docházelo k nesprávnému nastavení odsazení řádků takového výrazu. Po hlubším přezkoumání jsem zjistil, že tato chyba je perzistentní i u konverzí víceřádkových výrazů vstupů, ovšem výrazy příkazů *run / shell / log* byly serializovány korektně.

Po vyhodnocení testování bylo tedy třeba učinit opravy – ty se z většiny týkaly ošetření krajních vstupních hodnot, na nichž funkcionality daných komponent selhávaly. Chyba způsobující nesprávné odsazení víceřádkových výrazů při konverzi pipeline byla opravena tak, aby jednotlivé řádky výrazů byly odsazeny na správnou úroveň. Opětovné spuštění všech testů scénářů *I–IV* pak vykázalo úspěšnost 100 %.

Tabulka 5.1: Přehled testovacích případů testovacího scénáře I

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
1	Objekt stavu výstupů s prázdnou množinou výstupů	Vykreslení komponenty bez chyb	Ano
2	Stavový objekt: *null*		Chyba vykreslení – neošetřený vstup
	Stavový objekt: *undefined*		
	Stavový objekt: prázdný řetězec		
3	Hodnota textového pole: prázdný řetězec	Hodnota není přidána do seznamu výstupů	Ano
	Hodnota textového pole: *null*		
	Hodnota textového pole: *undefined*		
4	Textové pole: jednořádkový řetězec; počáteční stav: prázdný seznam výstupů	Hodnota je přidána do seznamu výstupů	Ano
	Textové pole: víceřádkový řetězec; počáteční stav: prázdný seznam výstupů		
	Textové pole: víceřádkový řetězec; počáteční stav: neprázdný seznam výstupů		
5	Index výstupu: -1; počáteční stav: neprázdný seznam výstupů	Seznam výstupů zůstane nezměněn	Ne – neošetřený vstup (dochází k odebírání výstupů)
	Index výstupu: *null*; počáteční stav: neprázdný seznam výstupů		
	Index výstupu: *undefined*; počáteční stav: neprázdný seznam výstupů		
	Index výstupu: 100 (nepřítomen v seznamu výstupů); počáteční stav: neprázdný seznam výstupů		Ano
6	Index výstupu: 1; počáteční stav: neprázdný seznam výstupů	Dojde k odebrání výstupu ze seznamu	Ano
	Index výstupu: 3; počáteční stav: neprázdný seznam výstupů		

5.3 Testování v rámci systému WHL

Podobně jako i při testování modelovacího nástroje jsem se rozhodl k testování komponent v systému WHL použít testovací framework. Pro platformu .NET existuje hned několik takových nástrojů – mezi ty nejpoužívanější patří mj. **xUnit**, **NUnit** či **MSTest** [27]. Všechny tyto nástroje poskytují vesměs stejné možnosti testování, liší se tedy především ve způsobu zápisu testů a případně i v tom, zda je daný nástroj podporován vývojovým prostředím jako Visual Studio. Především vzhledem k poslednímu kritériu jsem pro účely testování v rámci systému WHL zvolil nástroj MSTest.

Nástroj MSTest disponuje různými možnostmi pro tvorbu unit testů včetně parametrizace testů, paralelního spouštění testů či různých typů asercí výsledků. Kvůli své integraci do vývojového prostředí Visual Studio navíc ještě uživateli nabízí schopnost spouštět jednotlivé testy či testové sady a procházet výsledky testů v grafickém prostředí. Nástroj ovšem neobsahuje možnost tvorby mock objektů, tudíž je nutné využít externí knihovny. Pro tvorbu mock objektů jsem tedy použil knihovnu **Moq** [28], jež toto umožňuje prostřednictvím lambda výrazů či API **LINQ** (Language-Integrated Query), sloužící k efektivnímu zápisu datových dotazů.

5.3.1 Tvorba testů

Testová sada je v rámci nástroje MSTest definována třídou s anotačním atributem **TestClass**. Pro metody dané třídy jsou pak k dispozici různé atributy určující fáze životního cyklu testu či testové sady, v níž se daná metoda zavolá. Základním atributem je tak atribut **TestMethod** představující jeden konkrétní test. Parametrizovanou alternativu k tomuto atributu představuje **DataTestMethod**, pomocí něhož lze spolu s atributem **DataRow** daný test aplikovat na různé kombinace vstupních dat.

Struktura kódu testovací metody podobně jako u testů frameworku Jest (jejichž tvorba je popsaná v podsekci 5.2.1) na začátku obsahuje přípravu vstupních dat, mock objektů a dalších objektových instancí. Poté proběhne vyvolání testované funkcionality a na závěr aserce pro porovnání očekávaných hodnot a skutečných výsledků. Nástroj MSTest definuje také další atributy v rámci životního cyklu testu (**TestInitialize**, **TestCleanup**, **AssemblyInitialize** a **AssemblyCleanup**), jež jsou analogické k metodám frameworku Jest určeným ke stejnému účelu.

Tvorba a nastavení mock objektu pomocí knihovny Moq začíná vytvořením nové **Mock** instance, jejímž obecným parametrem je požadovaná třída a ostatní parametry jsou pak dosazeny jako argumenty odpovídajícího konstruktoru třídy. Následné nastavení chování mock objektu probíhá skrze volání jeho metody **Setup**, jejíž parametr má podobu lambda výrazu či LINQ výrazu. Dostupná nastavení chování mock objektu jsou porovnatelná s nastaveními mock objektů ve frameworku Jest.

5.3.2 Testovací scénáře

V systému WHL v rámci realizace řešení této diplomové práce vzniklo několik komponent, z nichž nejzásadnější jsou tyto tři: API controller **SnakemakeControllerAPI** pro obsluhu HTTP požadavků příchozích ze strany návrhového nástroje, služba **SnakemakeHandler** zajišťující operace nad uživatelským adresářem pro načítání či ukládání pipelines a API controller **SnakemakeJobControllerAPI**, který obstarává komunikaci s frontendem a s mock-up verzí middlewaru v rámci prototypu funkcionality spouštění pipelines. Pro každou z těchto komponent pak vznikl odpovídající testovací scénář. Rozvržení a obsah scénářů je podobný jako u scénářů *I–IV* (popsaných v podsekci 5.2.2) zabývajících se unit testováním modelovacího nástroje.

V následujících testovacích scénářích jsou tedy také zahrnuty případy, jež testují správnou funkčnost za běžných podmínek i ve výjimečných situacích. Níže lze nalézt výčet těchto tří scénářů, jež pro každý z nich obsahuje kromě popisu testované funkcionality i seznam dílčích testovacích případů. Souhrny spuštěných testovacích případů jsou opět zaznamenány v odpovídajících tabulkách; ty pak lze nalézt v příloze B pod příslušným označením. Zmiňované scénáře jsou tedy tyto:

- **Testovací scénář V:**
  - **testovaná funkcionalita**: zpracování HTTP požadavků příchozích ze strany modelovacího nástroje – controller **SnakemakeControllerAPI**
  - **tabulka testovacích případů**: tab. B.1 (viz příloha B)
  - **testovací případy**:
    - **Případ 22** – ověření funkcionality získání výčtu souborů uložených v uživatelském adresáři, pokud uživatel není přihlášen
    - **Případ 23** – ověření funkcionality získání výčtu souborů formátu JSON uložených v uživatelském adresáři, pokud uživatel není přihlášen
    - **Případ 24** – ověření funkcionality získání výčtu souborů uložených v uživatelském adresáři, pokud uživatel je přihlášen
    - **Případ 25** – ověření funkcionality získání výčtu souborů formátu JSON uložených v uživatelském adresáři, pokud uživatel je přihlášen
    - **Případ 26** – ověření funkcionality získání konkrétního souboru z uživatelského adresáře, pokud uživatel není přihlášen
    - **Případ 27** – ověření funkcionality získání konkrétního souboru z uživatelského adresáře, pokud soubor neexistuje
    - **Případ 28** – ověření funkcionality získání konkrétního souboru z uživatelského adresáře, pokud soubor existuje
    - **Případ 29** – ověření funkcionality uložení souboru do uživatelského adresáře, pokud uživatel není přihlášen

        - **Případ 30** – ověření funkcionality uložení souboru do uživatelského adresáře, pokud uživatel je přihlášen

- **Testovací scénář VI:**

    - **testovaná funkcionalita**: provádění souborových operací čtení či zápisu nad uživatelským adresářem – služba **SnakemakeHandler**
    - **tabulky testovacích případů**: tab. B.2 (viz příloha B)
    - **testovací případy**:
        - **Případ 31** – ověření funkcionality získání výčtu souborů uložených v uživatelském adresáři, pokud adresář neexistuje
        - **Případ 32** – ověření funkcionality získání výčtu souborů formátu JSON uložených v uživatelském adresáři, pokud adresář neexistuje
        - **Případ 33** – ověření funkcionality získání výčtu souborů uložených v uživatelském adresáři, pokud adresář existuje
        - **Případ 34** – ověření funkcionality získání výčtu souborů formátu JSON uložených v uživatelském adresáři, pokud adresář existuje
        - **Případ 35** – ověření funkcionality získání konkrétního souboru z uživatelského adresáře, pokud adresář neexistuje
        - **Případ 36** – ověření funkcionality získání konkrétního souboru z uživatelského adresáře, pokud soubor neexistuje
        - **Případ 37** – ověření funkcionality získání konkrétního souboru z uživatelského adresáře, pokud soubor existuje
        - **Případ 38** – ověření funkcionality uložení souboru do uživatelského adresáře, pokud adresář neexistuje
        - **Případ 39** – ověření funkcionality uložení souboru do uživatelského adresáře, pokud adresář existuje

- **Testovací scénář VII:**

    - **testovaná funkcionalita**: zpracování HTTP požadavků příchozích ze frontendu v rámci prototypu funkcionality spouštění pipelines – controller **SnakemakeJobControllerAPI**
    - **tabulka testovacích případů**: tab. B.3 a B.4 (viz příloha B)
    - **testovací případy**:
        - **Případ 40** – ověření funkcionality vytvoření a spuštění úlohy, pokud uživatel není přihlášen
        - **Případ 41** – ověření funkcionality zjištění stavu úlohy, pokud uživatel není přihlášen

- **Případ 42** – ověření funkcionality získání výčtu výstupních souborů úlohy, pokud uživatel není přihlášen
- **Případ 43** – ověření funkcionality získání výčtu pipelines uložených na serveru, pokud uživatel není přihlášen
- **Případ 44** – ověření funkcionality získání výčtu uživatelem nahraných souborů, pokud uživatel není přihlášen
- **Případ 45** – ověření funkcionality stažení vybraných výstupních souborů úlohy, pokud uživatel není přihlášen
- **Případ 46** – ověření funkcionality vytvoření úlohy a jejího spuštění končícího úspěchem
- **Případ 47** – ověření funkcionality vytvoření úlohy a jejího spuštění končícího neúspěchem
- **Případ 48** – ověření funkcionality zjištění stavu úlohy, kdy úloha není ve stavu "dokončená"
- **Případ 49** – ověření funkcionality zjištění stavu úlohy, kdy úloha je ve stavu "dokončená"
- **Případ 50** – ověření funkcionality získání výčtu výstupních souborů úlohy, pokud uživatel je přihlášen
- **Případ 51** – ověření funkcionality získání výčtu pipelines uložených na serveru, pokud uživatel je přihlášen
- **Případ 52** – ověření funkcionality získání výčtu uživatelem nahraných souborů, pokud uživatel je přihlášen
- **Případ 53** – ověření funkcionality stažení vybraných výstupních souborů úlohy, pokud uživatel je přihlášen

5.3.3 Vyhodnocení testování

Celková úspěšnost spuštěných testů testovacích scénářů *V–VII* byla téměř úplná. Jediné selhání testu nastalo u testovacího případu č. 27, v němž byla testována situace, při které došlo ke zpracování požadavku na získání neexistujícího souboru. Neošetření této situace zapříčinilo, že metoda controlleru **GetFile** vrátila odpověď s hodnotou souboru nastavenou na *null* místo očekávaného chybového HTTP kódu 404.

Tento nedostatek jsem tedy následně opravil tak, aby daná metoda controlleru skutečně ošetřovala situaci s požadavkem na neexistující soubor a případně pak vracela správný HTTP kód. Po opravě následovalo kontrolní spuštění testů, které již mělo plnou úspěšnost.

Kapitola 6

Závěr

Tato diplomová práce řešila problematiku vizuálního modelování výpočetních pipelines a jejím cílem bylo rozšířit webový systém WHL o nástroj pro modelování pipelines nástroje Snakemake. Za tímto účelem byl nejprve představen samotný nástroj Snakemake a poté byla provedena rešerše existujících nástrojů pro modelování pipelines, jež u několika takovýchto nástrojů rozebírala jejich účelnost a způsoby použití.

Následně byly analyzovány různé přístupy k návrhu a realizaci řešení, z nichž byly vybrány ty nejvhodnější. Pak byly shrnuty nejdůležitější technologie a nástroje použité k realizaci řešení. Návrhová sekce čtenáře informovala jak o návrhu samotné aplikace modelovacího nástroje, tak i o návrhu integrace tohoto nástroje do systému WHL. Návrh aplikace byl použit k implementaci řešení v podobě rozšíření webové komponenty Elsa Designer.

Výsledná webová komponenta byla integrována do systému WHL, ten však bylo ještě nutné implementačně rozšířit o funkcionalitu zajišťující komunikaci s modelovacím nástrojem a také o prototyp funkcionality spouštění pipelines skrze middleware. Po dokončení implementační části bylo provedeno testování řešení prostřednictvím unit testování na straně modelovacího nástroje i na straně systému WHL. Výsledky těchto testů byly vyhodnoceny a podle nich byla implementace řešení patřičně opravena.

Výsledné řešení je vypracováno do takového stavu, kdy je možno jej používat k praktickému modelování reálných Snakemake pipelines i většího rozsahu. Řešení skýtá přínos především pro bioinformatiky, kteří jsou již seznámeni s nástrojem Snakemake a používají jej k tvorbě výpočetních pipelines. Pro tyto uživatele poskytuje modelovací nástroj možnosti pro přehlednější tvorbu pipelines skrze jejich vizuální reprezentaci jako sadu bloků, jejichž vzájemné propojení představuje jednotlivé datové závislosti. Je ovšem nutno poznamenat, že použití nástroje stále předpokládá znalost jazyka Python a syntaxe Snakefile.

Modelovací nástroj je realizován jako webová komponenta s nízkým množstvím externích závislostí. V této diplomové práci byla integrace výsledného nástroje demonstrována za použití systému WHL, nástroj je ovšem možné nasadit do jakéhokoliv jiného webového prostředí, jež je podpořeno

webovým serverem. Ovšem vzhledem k tomu, že modelovací nástroj má podobu webové komponenty závislé na běhu na webovém serveru, není možno tento nástroj použít bez internetového připojení či nutnosti spouštět lokální webový server. Právě v tomto vidím jedno z omezení realizovaného řešení.

Tento aspekt řešení by se v rámci možného budoucího rozvoje nástroje dal vyřešit např. vsazením nástroje do frameworku Electron, což by vedlo k vytvoření nativní desktopové aplikace. Taktéž by bylo možno přetvořit nástroj po vzoru progresivních webových aplikací (PWA), což by ovšem již znemožňovalo využití mnoha nativních funkcionalit. Další omezení realizovaného řešení tkví v nekompletní nabídce možností pro modelování Snakemake pravidel. Současné řešení nabízí ty nejdůležitější z nich (nastavení vstupů, výstupů a příkazů *run*, *shell* a *log*), v budoucnu by ovšem bylo vhodné dopracovat i další možná nastavení (např. nastavení priority, počtu vláken či definice externích skriptů).

Další omezení se týká návrhového plátna nástroje, kde není možno upravovat šipky vazeb mezi bloky tak, aby uživatel mohl uspořádat vazby po svém za účelem lepší přehlednosti modelu. Toto omezení vyplývá z použití modelovací knihovny jsPlumb, v rámci budoucího rozvoje nástroje by tak stálo za zvážení, zda tuto knihovnu pro vykreslování bloků a vazeb nenahradit jinou knihovnou, jež bude podporovat více možností vizuálních úprav.

Integrace modelovacího nástroje do systému WHL v dosavadní podobě umožňuje pouze načítání a ukládání modelů pipeline či jejich Snakefile konverzí v rámci úložiště webového serveru a také spouštění namodelovaných pipelines v rámci prototypu této funkcionality. Součástí možného budoucího rozvoje by tak mohlo být rozšíření systému WHL tak, aby bylo možné spouštět a monitorovat úlohy namodelovaných pipelines v kombinaci s nahrazením použité mock-up verze middlewaru za rozhraní middlewaru HEAppE.

Literatura

1. KÖSTER, Johannes; RAHMANN, Sven. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*. 2012-08, roč. 28, č. 19, s. 2520–2522. ISSN 1367-4803. Dostupné z DOI: 10.1093/bioinformatics/bts480.
2. *Snakemake — Snakemake 5.32.0 documentation* [online] [cit. 2021-02-04]. Dostupné z: https://snakemake.readthedocs.io.
3. AL, Mölder F; Jablonski KP; Letcher B et. Sustainable data analysis with Snakemake [version 1; peer review: awaiting peer review]. *F1000Research*. 2021-01. Dostupné z DOI: 10.12688/f1000research.29032.1.
4. BERTHOLD, Michael R.; CEBRON, Nicolas; DILL, Fabian; GABRIEL, Thomas R.; KÖTTER, Tobias; MEINL, Thorsten; OHL, Peter; SIEB, Christoph; THIEL, Kilian; WISWEDEL, Bernd. KNIME: The Konstanz Information Miner. In: *Data Analysis , Machine Learning and Applications : Proceedings of the 31st Annual Conference of the Gesellschaft für Klassifikation e. V., Albert-Ludwigs-Universität Freiburg, March 7 9 , 2007*. New York: Springer, 2007.
5. *KNIME | Open for Innovation* [online] [cit. 2021-02-06]. Dostupné z: https://www.knime.com.
6. *Galaxy Community Hub* [online] [cit. 2021-02-07]. Dostupné z: https://galaxyproject.org.
7. AFGAN, Enis; BAKER, Dannon; BATUT, Bérénice; BEEK, Marius van den; BOUVIER, Dave; ČECH, Martin; CHILTON, John; CLEMENTS, Dave; CORAOR, Nate; GRÜNING, Björn A; GUERLER, Aysam; HILLMAN-JACKSON, Jennifer; HILTEMANN, Saskia; JALILI, Vahid; RASCHE, Helena; SORANZO, Nicola; GOECKS, Jeremy; TAYLOR, James; NEKRUTENKO, Anton; BLANKENBERG, Daniel. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Research*. 2018-05, roč. 46, č. W1, W537–W544. ISSN 0305-1048. Dostupné z DOI: 10.1093/nar/gky379.
8. *Galaxy* [online] [cit. 2021-02-07]. Dostupné z: https://usegalaxy.org.
9. *ELSA · An open source .NET Standard workflows library* [online] [cit. 2021-02-10]. Dostupné z: https://elsa-workflows.github.io/elsa-core/.

10. SCHOORSTRA, Sipke. Building Workflow Driven .NET Core Applications with Elsa. *Medium*. 2019-11. Dostupné také z: https://sipkeschoorstra.medium.com/building-workflow-driven-net-core-applications-with-elsa-139523aa4c50.

11. *HTML: HyperText Markup Language | MDN* [online] [cit. 2021-02-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTML.

12. *World Wide Web Consortium (W3C)* [online] [cit. 2021-02-20]. Dostupné z: https://www.w3.org.

13. *CSS: Cascading Style Sheets | MDN* [online] [cit. 2021-02-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS.

14. *CSS preprocessor - MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online] [cit. 2021-02-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor.

15. *JavaScript | MDN* [online] [cit. 2021-02-20]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

16. MILOJKOVIC, N.; GHAFARI, M.; NIERSTRASZ, O. It's Duck (Typing) Season! In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 2017, s. 312–315. Dostupné z DOI: 10.1109/ICPC.2017.10.

17. *TypeScript: Typed JavaScript at Any Scale* [online] [cit. 2021-02-21]. Dostupné z: https://www.typescriptlang.org.

18. *Run JavaScript Everywhere* [online] [cit. 2021-02-21]. Dostupné z: https://nodejs.dev.

19. *Stencil* [online] [cit. 2021-02-22]. Dostupné z: https://stenciljs.com.

20. *Web Components | MDN* [online] [cit. 2021-02-23]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Web_Components.

21. *Polyfill - MDN Web Docs Glossary: Definitions of Web-related terms | MDN* [online] [cit. 2021-02-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Polyfill.

22. *jsPlumb Toolkit Documentation* [online] [cit. 2021-02-25]. Dostupné z: https://docs.jsplumbtoolkit.com/community/current/index.html.

23. *A Tour of C Sharp - C Sharp Guide | Microsoft Docs* [online] [cit. 2021-03-03]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/.

24. *Introduction to ASP.NET Core | Microsoft Docs* [online] [cit. 2021-03-04]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0.

25. *Welcome to Flask — Flask Documentation (1.1.x)* [online] [cit. 2021-04-27]. Dostupné z: https://flask.palletsprojects.com/en/1.1.x/.

26. *Jest · Delightful JavaScript Testing* [online] [cit. 2021-03-25]. Dostupné z: `https://jestjs.io`.

27. *Testování v .NET - .NET | Microsoft Docs* [online] [cit. 2021-03-31]. Dostupné z: `https://docs.microsoft.com/cs-cz/dotnet/core/testing`.

28. *moq/moq4: Repo for managing Moq 4.x* [online] [cit. 2021-03-31]. Dostupné z: `https://github.com/Moq/moq4`.

Příloha A

Tabulky testovacích scénářů modelovacího nástroje

Tabulka A.1: Přehled testovacích případů testovacího scénáře II – část 1.

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
7	Relevantní aktivity: prázdný seznam; zvolené vstupy: prázdný seznam	Vykreslení komponenty bez chyb	Ano
	Relevantní aktivity: neprázdný seznam; zvolené vstupy: prázdný seznam		
	Relevantní aktivity: prázdný seznam; zvolené vstupy: neprázdný seznam		
8	Relevantní aktivity: *null*; zvolené vstupy: *null*		Chyba vykreslení – neošetřený vstup
	Relevantní aktivity: *undefined*; zvolené vstupy: *undefined*		
9	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: *null*; ID výstupu: *null*	Seznam zvolených vstupů zůstane nezměněn	Ne – neošetřený vstup (dochází k přidávání vstupů)
	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: *undefined*; ID výstupu: *undefined*		

Tabulka A.2: Přehled testovacích případů testovacího scénáře II – část 2.

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
10	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: 1; ID výstupu: 3	Daný vstup je přidán do seznamu	Ano
11	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: *null*; ID výstupu: *null*	Seznam zvolených vstupů zůstane nezměněn	Ano
	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: *undefined*; ID výstupu: *undefined*		
	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: 1; ID výstupu: 0 (nepřítomen ve výstupech aktivity)		
	Relevantní aktivity: neprázdný seznam; zvolené vstupy: prázdný seznam; ID aktivity: 0; ID výstupu: 0		
12	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: 0; ID výstupu: 0	Daný vstup je odebrán ze seznamu	Ano
	Relevantní aktivity: neprázdný seznam; zvolené vstupy: neprázdný seznam; ID aktivity: 2; ID výstupu: 2		

Tabulka A.3: Přehled testovacích případů testovacího scénáře III

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
13	Vstupní pipeline: *null*	Graf pipeline je označen jako neacyklický	Ne – selhání (neošetřený vstup)
	Vstupní pipeline: *undefined*		
14	Prázdná pipeline	Graf pipeline je označen jako acyklický	Ano
	Pipeline s jednou aktivitou		
	Acyklická pipeline (aktivity: {A, B, C}; vazby: {A –> B, A –> C})		
	Acyklická pipeline (aktivity: {A, B, C}; vazby: {A –> B, A –> C, B –> C})		
	Pipeline s dvěma acyklickými komponentami		
	Pipeline s dvěma nepropojenými aktivitami		
15	Cyklická pipeline (aktivity: {A, B, C}; vazby: {A –> B, B –> C, C –> A})	Graf pipeline je označen jako cyklický	Ano
	Rozšířená cyklická pipeline		
16	Vstupní pipeline: *null*	Graf pipeline je označen jako nesouvislý	Ne – selhání (neošetřený vstup)
	Vstupní pipeline: *undefined*		
17	Pipeline s jednou aktivitou	Graf pipeline je označen jako souvislý	Ano
	Souvislá pipeline (aktivity: {A, B, C}; vazby: {A –> B, A –> C})		
	Cyklická pipeline (aktivity: {A, B, C}; vazby: {A –> B, B –> C, C –> A})		
	Rozšířená souvislá pipeline		
	Rozšířená souvislá cyklická pipeline		
18	Prázdná pipeline	Graf pipeline je označen jako nesouvislý	Ano
	Pipeline s dvěma grafovými komponentami		
	Pipeline s dvěma nepropojenými aktivitami		

Tabulka A.4: Přehled testovacích případů testovacího scénáře IV – část 1.

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
19	Vstupní pipeline: *null*	Pipeline je konvertována jako prázdný řetězec	Ne – selhání (neošetřený vstup)
	Vstupní pipeline: *undefined*		
	Vstupní pipeline: Python blok s výrazem hodnoty *null*		Ne – vrácen neprázdný řetězec (neošetřený vstup)
	Vstupní pipeline: Python blok s výrazem hodnoty *undefined*		
20	Prázdná pipeline Pipeline s dvěma prázdnými aktivitami		Ano
	Pipeline s dvěma pojmenovanými prázdnými pravidly		
	Pipeline: dvě nepojmenované pravidla s validními vstupy a výstupy		
	Pipeline: neprázdné pojmenované pravidlo, blok vstupních dat s výrazem hodnoty prázdného řetězce		
	Pipeline: Python blok s výrazem hodnoty prázdného řetězce		

Tabulka A.5: Přehled testovacích případů testovacího scénáře IV – část 2.

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
21	Pipeline: neprázdné pojmenované collector pravidlo, blok vstupních dat s víceřádkovým výrazem	Pipeline je konvertována do odpovídajícího řetězce – nezáleží na pořadí pravidel (s výjimkou Python bloku a collector pravidla)	Ne – chyba serializace víceřádkového vstupu
	Pipeline: neprázdný Python blok		Ano
	Pipeline: neprázdné pojmenované collector pravidlo, dva validní bloky vstupních dat		
	Pipeline: neprázdné pojmenované pravidlo s příkazy *run*, *shell* a *log*, dva validní bloky vstupních dat		
	Pipeline: dvě neprázdná pojmenovaná pravidla s příkazy *run*, *shell* a *log*, dva validní bloky vstupních dat		
	Pipeline: neprázdné pojmenované pravidlo s příkazy *shell* a *log*, dva validní bloky vstupních dat		
	Pipeline: blok vstupních dat a dvě neprázdná pojmenovaná pravidla s příkazy *run*, *shell* a *log* (aktivity jsou propojeny postupně)		
	Pipeline: blok vstupních dat a neprázdné pojmenované pravidlo s příkazy *run*, *shell* a *log*, neprázdné collector pravidlo (aktivity jsou propojeny postupně)		
	Pipeline: kompletní pipeline A se všemi typy prvků		
	Pipeline: kompletní pipeline B se všemi typy prvků		

Příloha B

Tabulky testovacích scénářů systému WHL

Tabulka B.1: Přehled testovacích případů testovacího scénáře V

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
22	Výchozí stav: uživatel není přihlášen	HTTP kód 401	Ano
23			
24	Výchozí stav: uživatel je přihlášen	Výsledek je odpověď typu JSON	Ano
25			
26	Výchozí stav: uživatel není přihlášen	HTTP kód 401	Ano
27	Výchozí stav: uživatel je přihlášen, požadovaný soubor neexistuje	HTTP kód 404	Ne – vrácen *null* soubor (neošetřený vstup)
28	Výchozí stav: uživatel je přihlášen, požadovaný soubor existuje	Výsledek je soubor typu JSON	Ano
29	Výchozí stav: uživatel není přihlášen	HTTP kód 401	Ano
30	Výchozí stav: uživatel je přihlášen	HTTP kód 200	Ano

Tabulka B.2: Přehled testovacích případů testovacího scénáře VI

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
31	Výchozí stav: uživatelský adresář neexistuje	Prázdný seznam; adresář je vytvořen	Ano
32			
33	Výchozí stav: uživatelský adresář existuje a je neprázdný	Seznam názvů souborů v uživatelském adresáři	Ano
34		Seznam názvů JSON souborů v uživatelském adresáři	
35	Výchozí stav: uživatelský adresář neexistuje	Soubor hodnoty *null*; adresář je vytvořen	Ano
36	Výchozí stav: uživatelský adresář existuje, požadovaný soubor neexistuje	Soubor hodnoty *null*	Ano
37	Výchozí stav: uživatelský adresář i požadovaný soubor existují	Soubor není hodnoty *null*	Ano
38	Neprázdný řetězec; výchozí stav: uživatelský adresář neexistuje	Adresář i soubor jsou vytvořeny, obsah souboru odpovídá vstupnímu řetězci	Ano
39	Neprázdný řetězec; výchozí stav: uživatelský adresář existuje	Soubor je vytvořen a jeho obsah odpovídá vstupnímu řetězci	Ano

Tabulka B.3: Přehled testovacích případů testovacího scénáře VII – část 1.

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
40	Výchozí stav: uživatel není přihlášen	HTTP kód 401	Ano
41			
42			
43			
44			
45			
46	Seznam názvů vstupních souborů, název pipeline; založení úlohy končí úspěchem	Volání middleware API: autentizace, vytvoření úlohy, získání metody dat. přenosu, spuštění úlohy; specifikace úlohy obsahuje korektní parametry; odpověď je typu JSON a stav úlohy je "konfigurovaná"	Ano
47	Seznam názvů vstupních souborů, název pipeline; založení úlohy končí neúspěchem	Volání middleware API: autentizace, vytvoření úlohy; specifikace úlohy obsahuje korektní parametry; odpověď je typu JSON a stav úlohy je "zrušená"	Ano
48	Stav úlohy bude "spuštěná"	Volání middleware API: autentizace, zjištění stavu úlohy; odpověď je typu JSON	Ano

Tabulka B.4: Přehled testovacích případů testovacího scénáře VII – část 2.

ID případu	Vstupní data	Očekávaný výsledek	Úspěch
49	Stav úlohy bude "dokončená"	Volání middleware API: autentizace, zjištění stavu úlohy, získání metody dat. přenosu; odpověď je typu JSON	Ano
50	Výchozí stav: uživatel je přihlášen	Výsledek je odpověď typu JSON	Ano
51			
52			
53	Výchozí stav: požadovaný soubor existuje	Výsledek je odpověď typu "*application/octet-stream*"	Ano

Příloha C

Obsah souboru elektronické přílohy

/
- **readme.txt** - soubor s popisem obsahu elektronické přílohy
- **SnakemakeDesignerUnitTests** - adresář se zdrojovými kódy testů na straně systému Web HPC Launcher
- **snakemake-designer** - adresář se zdrojovými kódy webové komponenty nástroje Snakemake Designer včetně kódů testů na straně nástroje
- **snakemake-middleware-mock** - adresář se zdrojovými kódy mock-up verze rozhraní middlewaru
- **thesis** - adresář se zdrojovými kódy Latex verze textu diplomové práce
- **WebHPCLauncher** - adresář se zdrojovými kódy systému Web HPC Launcher včetně výsledné kompilované podoby nástroje Snakemake Designer